Department of Computer Science and Software Engineering
University of Canterbury

January 2009

A Constraint-Based ITS for the Java Programming Language

A thesis submitted in partial fulfilment of the requirements
for the Degree of Master of Science in Computer Science
in the University of Canterbury by

Jay Holland

Supervisor: Professor Dr. Antonija Mitrović
Associate Supervisor: Dr. Brent Martin
Examiner: Dr. Peter Andreae[1]

[1] Victoria University

To my family

Abstract

Programming is one of the core skills required by Computer Science undergraduates in tertiary institutions worldwide, whether for study itself, or to be used as a tool to explore other relevant areas. Unfortunately, programming can be incredibly difficult; this is for several reasons, including the youth, depth, and variety of the field, as well as the youth of the technology that frames it. It can be especially problematic for computing neophytes, with some students repeating programming courses not due to academic laziness, but due to an inability to grasp the core concepts. The research outlined by this thesis focuses on our proposed solution to this problem, a constraint-based intelligent tutoring system for teaching the Java programming language, named J-LATTE.

J-LATTE (Java Language Acquisition Tile Tutoring Environment) is designed to solve this problem by providing a problem-solving environment for students to work through programming problems. This environment is unique in that it partitions interaction into a concept mode and a coding mode. Concept mode allows the student to form solutions using high-level Java concepts (in the form of tiles), and coding mode allows the student to enter Java code into these tiles to form a complete Java program. The student can, at any time, ask for feedback on a solution or partial solution that they have formed.

A pilot study and two full evaluations were carried out to test the effectiveness of the system. The pilot study was run with an assignment given to a postgraduate Computer Science course, and because of the advanced knowledge level of the students, it was not designed to test teaching effectiveness, but instead was useful in determining usability issues and identifying any software errors.

The full evaluations of the system were designed to give insight into the teaching effectiveness of J-LATTE, by comparing the results of using the system against a simulated classroom situation. Unfortunately, the participant base was small, for several reasons that are explained in the thesis. However, the results prove interesting otherwise and for the most part are positive towards the effectiveness of J-LATTE. The participants' knowledge did improve while interacting with the system, and the subjective data collected shows that students like the interaction style and value the feedback obtained.

Contents

Abstract i

List of Figures vii

List of Tables ix

Acknowledgements x

Chapter 1 Introduction 1

1.1 Intelligent Tutoring Systems 2

1.2 The Difficulty of Learning Programming 3

1.3 J-LATTE: Teaching Java with Constraints 4

1.4 Thesis Outline 5

Chapter 2 Intelligent Tutoring Systems 6

2.1 Architecture 8

2.2 Student Modelling 10

2.2.1 Model-Tracing 11

2.2.2 Constraint-Based Modelling 13

Chapter 3 Programming 16

3.1 The Java Programming Language 17

3.2 Important Programming Constructs 18
3.3 Programming Environments 23
3.3.1 Text Editors 24
3.3.2 Programming Editors 24
3.3.3 IDEs 26
3.4 Intelligent Programming Tutors 29
3.5 Discussion 35

Chapter 4 Design **39**
4.1 Approach 40
4.2 Curriculum 45
4.3 Problem Design 46
4.4 Constraint Design 49
4.4.1 Syntactic Constraints 51
4.4.2 Style Constraints 51
4.5 Verifying Semantic Correctness 52
4.5.1 Ideal Solutions 55
4.5.2 Semantic Constraints 56

Chapter 5 Implementation **57**
5.1 Architecture 57
5.2 Problem Solving Interface 58
5.2.1 Loading a Problem 60
5.2.2 Solution Formation 61
5.2.3 Solution Submission 65
5.2.4 User Interaction Process Summary 67
5.3 Client-Server Overview 68

5.4 Client 69
5.4.1 Internal Student Solution Representation 69
5.4.2 Tile Manipulation Implementation 71
5.4.3 Solution Submission 72
5.5 Server 73
5.5.1 Problem Representation 74
5.5.2 Student Solution Representation 78
5.5.3 Constraint Representation 81
5.5.4 Solution Evaluation 84
5.5.5 Student Modeller 85

Chapter 6 Evaluation 87

6.1 Pilot Study 87
6.1.1 Results 88
6.2 Evaluation Study 1 (August 2007) 89
6.2.1 Recruitment of Participants 91
6.2.2 Experimental Design 92
6.2.3 Results and Analysis 95
6.3 Evaluation Study 2 (August 2008) 98
6.3.1 Experimental Design 99
6.3.2 Procedure 100
6.3.3 Results and Analysis 101

Chapter 7 Conclusions 105

7.1 J-LATTE 105
7.2 Evaluation 106
7.3 Further Work 108

References **110**

Appendix A **Information Sheet** **118**

Appendix B **Consent Form** **120**

Appendix C **Pre and Post Tests** **122**

Appendix D **Questionnaire** **127**

Appendix E **NZCSRSC 2007 Paper** **129**

List of Figures

2.1 A Typical ITS Architecture 8
2.2 Model-tracing production rule example 12
2.3 CBM constraint examples for the SQL domain 14

3.1 Java Source Code: plain vs syntax coloured and indented 24
3.2 Interface of BlueJ 28
3.3 GREATERP production rule example 31

4.1 Example of an algorithm represented in pseudo-code 43
4.2 Java domain ontology, as developed in CAS 50
4.3 Syntactic constraint design: from grammar to natural-language constraint 51
4.4 Example of multiple valid solutions to a single problem 53
4.5 Semantic constraint example 56

5.1 System Architecture of J-LATTE 58
5.2 J-LATTE Interface Layout (Coding Mode) 59
5.3 Interface - Problem Description 60
5.4 Solution Workspace (initial state) 63
5.5 Solution Workspace (with formed solution) 63
5.6 Interface - Tile Pane 64

5.7 Interface - Dragging Action. 64
5.8 Interface - Context 65
5.9 Interface - Feedback Pane 67
5.10 Complete solution for “Predicate” problem in Listing 5.1 72
5.11 Complete solution for “Printing” problem 74
5.12 Complete solution for “Iteration” problem 75
5.13 Ideal Solution Grammar 79

6.1 Probability of violating a constraint for the 2007 evaluation 97
6.2 Probability of violating a constraint for the 2008 evaluation 103

List of Tables

6.1 Evaluation 2007: System interaction statistics 95
6.2 Evaluation 2007: Pretest and posttest scores 96
6.3 Evaluation 2007: Mean scores from the questionnaire 98
6.4 Evaluation 2008: System interaction statistics 101
6.5 Evaluation 2008: Pretest and posttest scores 103
6.6 Evaluation 2008: Mean scores from the questionnaire 104

Acknowledgements

Where does one begin? The start (preferably). Better make yourself a cup of coffee, this could take a while.

Firstly, a broad, sweeping, inclusive statement, to which anyone who says "You left me out!" can be referred to: I would like to thank my family and friends, for their support and for putting up with my absence. Excellent! Thanks to my parents, for understanding that "I'm actually quite busy" means I'm actually quite busy. To my long-term housemates, Sean and Tanya, for precisely the same reason (but with having even more patience).

To my research group, ICTG, for all the discussions and feedback. To my partners in (MSc.) crime, Kon Zakharov and Nancy Milik, with whom I now have a large repertoire of inside jokes that nobody else will ever understand, no matter how well we explain them or how many cats we free. To Moffat Mathews, for encouragement, kindness, and showing me the meaning of Christmas. To my associate supervisor, Dr. Brent Martin, whose comments and questions helped make this thesis that-much-better. To everyone else in the group, in no particular order[1], Amanda, Pramudi, Amali, Sharon, David, Aidan, Nilufar, Devon, Martin, and Linda, you guys have increased the fun value by an order of magnitude (maybe more, but my data collection on this issue was a bit imprecise).

[1] Apart from etymological age.

To the academic and general staff of CSSE at the University of Canterbury, thanks for creating a great place to do this kind of research. Gillian, Alex, Phil and the programmers, you keep the CSSE world turning! To my many current and former colleagues in CSSE, for friendship and lunchtime hijinks[2] - Jason, Ray, Delio, Sascha, Andrew(s), Jung, Adrian, Xianglin, Annabel, Taher, Mohammad, Rana, Oliver(s), Amadeus, Rowan, Neha, Tae, The Chia, Pubudini, Eri, Marissa, Sum, Ki, Simon, Kieran, Jarrod, Keji, Ahmed, Marco, Asmeet, and many, many others.

To Nancy (*two* mentions? You lucky thing), who, with the powers of persuasion that we all know, encouraged me to embark on the adventure that is postgraduate study (and yes, I am thanking you for this!).

And finally, my greatest gratitude[3] goes to my supervisor, Professor Tanja Mitrović. There are no words (not even these!) that can express how valuable your guidance and encouragement has been over the last four years, as well as your patience. No matter how busy you have been, you have always had time to address my (and all of our) many questions, and your generosity in so many ways towards us has created something that is more akin to a family than a group. The forthcoming nomination for sainthood is just a small yet insufficient token of our appreciation. Хвала лепо!

[2] Hijinks = eating lunch and discussing Slashdot. We're wild and crazy guys!

[3] Anyone who has been supervised by her will certainly understand.

CHAPTER 1

Introduction

Acquisition of computer programming skill is a core component of the Computer Science curriculum, a fact reflected by the many first-year tertiary prescriptions that require a student to undertake some kind of programming course. There are many aspects to programming theory, such as program control-flow and scope, and this variety can make it difficult for students already lacking a suitable information technology background. It is generally accepted that the best way to introduce these ideas is through the teaching of a specific language. The Java programming language provides an appropriate introductory programming syllabus. Due to its low-level abstractions and system-independent nature, the student is able to concentrate more on the general programming concepts rather than system idiosyncrasies.

Although programming courses tend to have material taught in lectures, most of the learning reinforcement takes place in laboratories, where practical tasks are carried out. An increasingly popular and effective way of improving student learning is through Intelligent Tutoring Systems (ITSs), which enhance learning by providing feedback personalised to a student. These have been shown to be effective for many different

disciplines and areas, including mathematics [Koedinger et al., 1997], physics [Gertner and VanLehn, 2000], and database design [Suraweera and Mitrovic, 2004].

The J-LATTE (Java Language Acquisition Tile Tutoring Environment) system is our attempt to teach the Java language to students, through a tutor that utilises the constraint-based modelling (CBM) methodology [Ohlsson, 1994]. By using constraints, we do not have to restrict the student to a set path while forming a solution, and with a constraint set that sufficiently covers the domain of Java programming, any solution that is valid for the domain is valid for the system. The only exception to this is that the system, in the interest of teaching good practice as well as syntactic and semantic correctness, also contains constraints that enforce good style (although these are not overly restrictive).

1.1 Intelligent Tutoring Systems

The ideal tutoring situation, in terms of learning effectiveness for a student, is one tutor to one student; it has been shown that these circumstances can have a significant (2 s.d) learning improvement over traditional classroom instruction [Bloom, 1984]. Unfortunately, for courses with many students it is not feasible to have one tutor per student due to both the lack of tutors with expertise and the economics involved in supplying each student with a tutor.

One modern solution to this problem is the ITS. ITSs are learning environments that have a goal of achieving tutoring effectiveness close to that of a human tutor. Using research from such diverse areas as artificial intelligence, education, and psychology, ITSs allow computers to be used as learning tools that come close to achieving this goal, which was impossible with earlier computer-based instruction. The core concepts of ITSs include reasoning about solutions to exercises, giving fine-grained feedback to

these solutions, keeping a model of the student's knowledge of domain concepts, and using this student model to adapt instruction to suit the student's knowledge.

1.2 The Difficulty of Learning Programming

Unfortunately, given the relative youth of the field compared to other disciplines, the important ideas unique to Computer Science, such as programming, are not widely disseminated into the wider human culture; due to this, and several other reasons, people new to the area find many of the concepts both difficult and alien. This is especially true of people who have not grown up with computers, and are having to simultaneously understand both the general concept of computing as well as the foreign nature of programming. The fact that most of the related technology is relatively fresh means that for many computing neophytes it's not just a case of learning how to program - to even get to the useful stage where programming knowledge is acquired, software must be installed, compilers must be configured etc. The depth and variety of the field are also large obstacles; there are several different types of programming languages, styles, and methodologies. Just because you can write a simple program doesn't mean your skills will scale up to larger and more complex tasks, especially if your understanding of the nature of programming is shallow. Also, programming is a complex process; it is not about acquiring facts and figures - to learn to program you must acquire *new cognitive skills*.

Another large difficulty with learning programming is that the problem is two-part; students have to learn how to design programs, as well as learn the language in which to write them. Often this is done all at once, as it is a 'chicken and egg' situation - design theory without the practical side leaves out the relevant context that gives a student a greater understanding, whereas a practical curriculum without the design side leads a

student to know what the 'tools' are, but are unable to combine them in a meaningful order. Therefore, learning program design and learning a language itself tend to be taught in a tightly-coupled way; the downside to this is that the student is overwhelmed with information. This is especially true with syntax in programming languages; a student may have a good grasp of the design aspects of programming, but is unaware of the fact because their program will not compile due to syntax errors. We believe that splitting design from coding will make learning easier.

1.3 J-LATTE: Teaching Java with Constraints

J-LATTE is our solution to tutoring Java programming. The system takes a novel approach to handling the two-part problem of teaching both language and design, whilst reducing transfer problems. The student must form solutions to various programming problems; for this process, we abstract out the statement-level and block-level concepts (such as assignments, loops, and conditionals), so that the student does not need to work immediately with atomic Java elements such as keywords and semi-colons if they do not feel comfortable with them, but can operate at that level whenever they are ready.

The environment that the system presents to the student is designed to reduce transfer problems while at the same time being easy to use. The final product of any solution the student submits will be a code listing, that will be compatible (with boilerplate code added) with any Java compiler. The abstracted concepts are represented by tiles, each containing a field that a student can complete with actual Java code. These tiles can be inserted and repositioned inside the student's solution, until the student is happy with the program they have created.

Another reason for our approach is that an ITS in an unconstrained environment (for example, with raw code) would encounter difficulties when it came to diagnosing

solutions, because programming is very open-ended. In our approach the student is defining explicitly some of the higher meaning, therefore aiding the program during solution evaluation, which in turn allows the system to deliver better feedback.

1.4 Thesis Outline

In Chapters 2 and 3, we discuss the background of our research, covering Intelligent Tutoring Systems, Constraint-Based Modelling, and programming. In Chapters 4 and 5, we talk about our solution to the given problem, presenting the design decisions and the implementation details of our system. In Chapter 6 we discuss the evaluation of the system, and present the results of this evaluation. Finally, in Chapter 7, we draw conclusions about our solution from the evaluation results.

CHAPTER 2

Intelligent Tutoring Systems

Personal tutoring is one of the most effective ways of enhancing learning. Due to growing populations, resource constraints, and the complexities of some fields, personal tutors are not always readily available, whereas computers are becoming more and more commonplace. From the early days of computing, Computer-Aided Instruction (CAI) and Computer-Based Instruction (CBI) have been prominent fields for research into how to achieve the same effectiveness as a personal human tutor (which often improve students' scores over conventional classroom instruction by 2 standard deviations [Bloom, 1984]). CBI and CAI supplement classroom instruction by delivering instructional material to the student via a computer [Taylor, 1980]. The instructional materials generally consist of selected readings, tutorials, or problems for drill and practice.

The first CAI systems were primitive in terms of how they reacted to the students' behaviour; there was little or no adaptation to the student's progress, and they generally just followed a script. The curriculum usually dictated the sequence of concepts or problems in the script and each student tracked the same path through the materials provided. This changed with the advent of ITSs (Intelligent Tutoring Systems).

ITSs are computer-based tutoring systems. Unlike earlier generations of teaching systems, they contain explicitly represented domain knowledge. This domain knowledge might be, for example, in the form of domain principles (called constraints) or procedural steps (called production rules). This allows ITSs to reason "intelligently" about the particular domain and provide the student with useful feedback at an appropriate level of granularity. ITSs also calculate the proficiency of students in various concepts related to the field of the tutor, and use this information to personalise tutoring. This skill-tracking is known as *student modelling*, and is described in Section 2.2 in detail. An interdisciplinary field, ITS theory draws from psychology, linguistics, artificial intelligence (AI) and education as well as computer science, as we try to model and understand the cognitive processes.

ITSs supplement classroom instruction by mimicking the actions of a personal tutor. Research in psychology and education has shown that each student is different in the way they learn; this includes their learning styles, their rate of learning a particular concept, and their ability to link concepts in a domain. While older generation systems were static in their delivery of content, ITSs are dynamic, enabling students to work at a level appropriate to their current knowledge. Instead of receiving a general message on submission of a particular solution, students receive feedback that is tailored to their current knowledge level and context. This allows the student to recognise their errors and learn from them [Ohlsson, 1996]. Feedback can also be scaffolded depending on expertise, allowing novices to be guided and instructed more than experts. The choice of material to deliver can be dynamic and individualised, based on suitability to the student's current knowledge state and history.

ITSs are becoming increasingly popular owing to their effectiveness (students using state-of-the-art tutors can score around 1 standard deviation higher than students in a conventional classroom [Anderson et al., 1995]). In one study, high school students

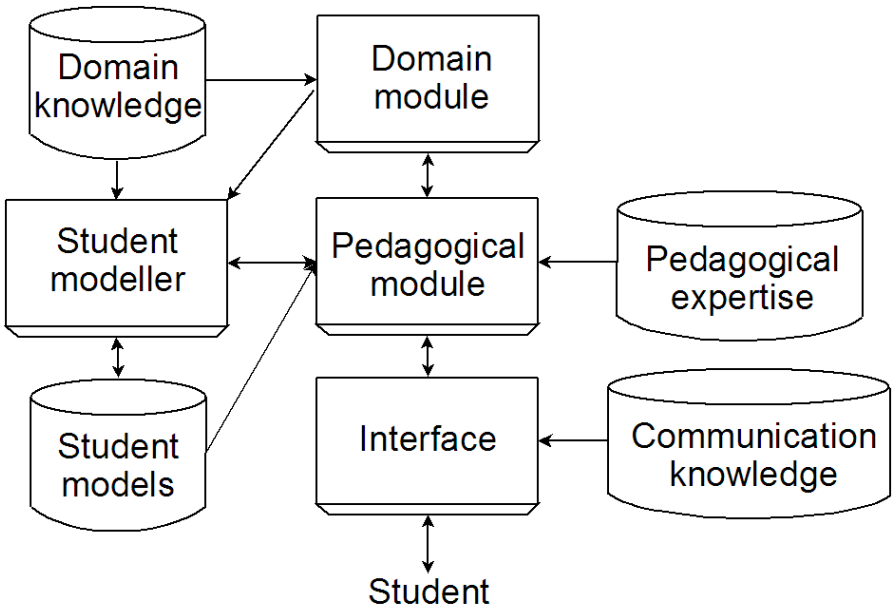

Figure 2.1: A Typical ITS Architecture

became so motivated that they went to extreme measures, such as skipping other classes, in order to use a particular system [Anderson et al., 1995]. ITSs have many advantages over regular CAI and 'solo' learning: the time taken for a student to learn is reduced, students may be able to acquire skills otherwise impossible learning 'on your own', and there is generally a greater understanding of the material.

2.1 Architecture

ITS architecture design has been specifically researched, with an ideal prototype being proposed [Beck et al., 1996]. Each ITS can be thought of being composed of several distinct parts, which interact in various ways. A typical architecture is shown in Figure 2.1. The modularity of this architecture theoretically allows for switching out components for different versions. The following is a description of the components of an ITS.

The domain module provides an interface to information in the domain knowledge store, where knowledge about the tutor's domain is stored (such as production rules or constraints, problems/exercises, and solutions). In some ITSs, the domain module is capable of solving problems.

The student modeller records information about the student's current state of knowledge, and stores the information for each student in a student model. It also handles evaluations of student solutions against the domain-model. The student modelling process is described in Section 2.2.

The pedagogical module (PM) is responsible for all the teaching decisions made in the ITS. Decisions are based on information gathered from each student's student model. The PM contains teaching strategies to maximise learning. These strategies vary between ITSs and are dependent on educational and learning theories.

Depending on the particular ITS implementation, the PM might decide on when a particular solution should be evaluated. One such strategy might leave the request for evaluation in the hands of the student, while another strategy might force an evaluation at particular points in the process (e.g. after each step). Some tutors also use a combination of both.

Problem selection strategies decide on how to select the 'next best problem' for a student [Mitrovic and Martin, 2003]. Feedback strategies select the level and type of feedback presented. They also determine the amount of feedback presented; for instance, feedback can be scaffolded depending on expertise. Metacognitive strategies select the type of metacognitive instruction to be presented. For example, self-explanation strategies might ask the student to explain why they undertook a certain action [Weerasinghe and Mitrovic, 2002]. Strategies can also be implemented to cope with various help-seeking behaviours [Aleven and Koedinger, 2000] and gaming [Baker et al., 2005].

Strategies that detect and encourage collaboration in collaborative tutors can also be implemented [Baghaei and Mitrovic, 2006].

Finally, the interface of an ITS allows the student to interact with the system, for example retrieving problems to work on, and entering and submitting solutions. A good ITS interface is designed such that it reduces the cognitive load on the student, so that the student can focus on solving the problem rather than trying to understand the representation. An ITS interface can also reduce the working memory load by providing glossaries to concepts in the domain.

2.2 Student Modelling

Student modelling is the process of describing the progress of students on concepts in a given domain, as an attempt to model each student's cognitive process. By analysing a student's results from previous problems and tests, a student model can be generated, describing how well a student knows each relevant topic. This information can be used to select the pedagogical method and material to tutor. Due to the amount of knowledge needed to be obtained to generate a comprehensive student model, it is agreed to be an intractable problem [Self, 1990]; research though has shown that a student model need not be complete or accurate to be practical.

A popular method for student modelling is model-tracing [Anderson et al., 1995], where problems are arranged procedurally with goals and subgoals, and the series of correct and incorrect procedures are recorded. A new paradigm, constraint-based modelling [Ohlsson, 1994], has recently emerged as a way of overcoming some of the disadvantages of these established systems. Other student modelling approaches exist [Greer and McCalla, 1994], but model-tracing and constraint-based modelling are the most prominent methods. We now describe and compare these two approaches.

2.2.1 Model-Tracing

Model-tracing (MT) is a method of student modelling based on the ACT-R (Adaptive Character of Thought-Rational) theory of learning [Anderson, 1996]. This theory states that goal-independent declarative knowledge (knowing *what* something is) is learnt through observation and instruction; declarative knowledge is later converted into goal-oriented production rules, which represent procedural knowledge (knowing *how* to do something). Each production rule can represent part of a domain model; hence a *set* of production rules can provide varying degrees of coverage of a domain.

This theory has been used to generate eight principles for how to design tutors employing MT, known as *cognitive tutors*. The first and main principle, related to student modelling, is that as a domain model can be represented by a production rule set, the student's competence (i.e. student model) can be represented by the union of a) the student's current correct knowledge, represented by a subset of the domain's production rules, and b) the student's *erroneous* knowledge, represented by a set of *buggy rules*. The production rules represent a path or series of paths from the initial empty solution state to the completed solution state, whereas the buggy rules represent diversions from these paths, which represent possible yet incorrect actions in the context of this domain.

A core feature of cognitive tutors is that the system's domain module can generate, step-by-step, how to complete a problem. Therefore, interaction with such an ITS is distinguished by immediate evaluation, and hence feedback, for each action that a student enters. If a student enters an incorrect action, then the tutor will inform the student that it was incorrect and why, and revert the solution back to after the last valid production rule. The system restricts the student from moving off the set path, meaning the system is in complete control of the possible solutions.

Although a cognitive tutor can still identify erroneous production rules by their absence from the domain model, a bug library must be created in order to generate useful feedback for when an incorrect action occurs (otherwise the feedback would be vague, i.e. "This action is incorrect"). Any incorrect production rule off a valid path is a candidate for the bug library; hence a bug library can easily become very large.

Each production rule is written as an "If...Then..." pair. The "If" part of the rule contains a goal and/or a state, and the "Then" part contains an action and/or setting of a new goal or state. As an example, Figure 2.2 shows a possible production rule for the algebra domain.

```
IF the goal is to solve x/a = c for x
THEN rewrite the equation as x=a.c
```

Figure 2.2: Model-tracing production rule example

There have been several successful cognitive tutors developed, covering a wide range of topics, including Algebra [Koedinger et al., 1997] and LISP programming [Anderson and Reiser, 1985]. In one study, students who used the PAT Algebra Tutor scored 1 standard deviation higher on the curriculum's target tests than non-PAT users.

Despite their success, cognitive tutors have several disadvantages, especially during development. Because all paths through a problem have to be explicitly programmed, the only strategies that a student can use are ones that have been thought of by the developers. If a domain has been poorly defined, then a student may find that his chosen path is rejected by the tutor, and therefore they will only be able to learn pre-specified problem-solving strategies. Even if the domain is well-defined it is still likely to be incomplete, which could mean that left-field "genius" strategies are unlikely to be accepted by the system. Developing a production-rule set that gives sufficient domain coverage is also time-consuming (100 development hours per 1 hour of instruction [Woolf

and Cunningham, 1987]). The development time is also exacerbated by the need to also develop a library of erroneous knowledge via the bug library.

2.2.2 Constraint-Based Modelling

Although MT tutors have been proven to be effective, the disadvantages mentioned show that this method of student modelling has several limitations. Constraint-based Modelling (CBM) provides a different way of handling the student modelling problem; it represents all declarative domain knowledge in the form of state constraints. Each constraint is an ordered pair made up of a relevance condition and a satisfaction condition, where each condition checks the state of a solution. A constraint can be stated in natural language as an "If...Then..." statement, i.e. "If condition x is true, then (for this constraint to be satisfied) it must be the case that condition y is also true".

Evaluation of solutions is as follows: for a given solution, the solution state is checked against all relevance conditions. Any relevant constraints (i.e. constraints whose relevance conditions are met by the student's solution) must be satisfied to have the solution be evaluated as correct - if the satisfaction condition of a relevant constraint fails, the constraint is violated. Any constraint violations indicate errors in the solution, therefore evaluation gives the system a list of errors in the solution. The history of each relevant constraint for a solution is recorded in the student model, which allows the system to gauge the student's knowledge, both in terms of correct understanding (satisfied constraints) and incorrect understanding (violated constraints) of domain knowledge.

All constraints within a particular domain represent a coverage, possibly incomplete, of that domain. Within a domain, constraints can be categorised into syntactic or semantic constraints. Syntactic constraints encode problem-independent syntactic (structural) knowledge; an example of this knowledge within the domain of SQL is that the NOT

operator, when used in the HAVING clause, must only be applied to conditions. This contrasts with semantic constraints, which encode problem-dependent semantic knowledge; an example of this knowledge within SQL is that you must name the tables where information is required from in the FROM clause. Often these semantic constraints use information from the problem statement or an ideal solution and compare it with the student's solution.

Figure 2.3 shows 2 pseudocode equivalents of constraints implemented by the constraint-based ITS SQL-Tutor [Mitrovic and Ohlsson, 1999], representing the previously stated declarative SQL knowledge.

Syntax constraint example:

Relevance: `IF the WHERE clause is not empty, and the HAVING clause contains the NOT operator`
Satisfaction: `THEN it must be the case that the object NOT is operating on is a condition`

Semantic constraint example:

Relevance: `IF the SELECT statement is not empty`
Satisfaction: `THEN it must be the case that it contains references to all tables that the problem statement requests you retrieve information from`

Figure 2.3: CBM constraint examples for the SQL domain

CBM is growing in popularity due to its many advantages over other methods, such as its diminished computational complexity due to the fact that the evaluation is essentially simple pattern-matching, as well as constraints being easy to write [Mitrovic et al., 2003]. Also, as constraint-based tutors do not require a bug library to be engineered, development time is shorter, as much development effort is taken up with discovering bugs in other types of knowledge modelling approaches. A problem solver is also not

required for CBM, which is advantageous as they are not appropriate for some domains, and difficult to develop for others; a problem solver can be used though if one can be developed. CBM also disassociates the domain knowledge from the problem-solving strategy, allowing a student to choose their own path to the solution.

Several successful constraint-based tutors have already been developed. SQL-Tutor [Mitrovic and Ohlsson, 1999] is a system that teaches the SQL database language, aimed at tertiary-level students in database design courses. It has a domain knowledge base of 700 constraints, and an interface designed with emphasis on reducing cognitive load. Several evaluations of this system have shown that students who use the system perform significantly better than students who do not [Mitrovic and Ohlsson, 1999]. Another successful system is the Language Builder ITS (LBITS) [Martin and Mitrovic, 2002], a tutor for teaching basic English skills to school children at elementary and secondary levels. Despite its short development time, an evaluation has shown it to be very effective, both in performance and motivation. Constraint-based tutors have also been designed for domains with representations other than text. ER-Tutor [Suraweera and Mitrovic, 2004] and Collect-UML [Baghaei et al., 2005] [Baghaei et al., 2007] are ITSs for tutoring ER modelling and UML software modelling respectively. Both of these domains employ a diagram-based representation.

CHAPTER 3

Programming

What is computer programming? At its most general, computer programming is creating a set of instructions that a computer can understand so that a task can be performed by the computer. Of course, the full definition is more complex; often the computer cannot directly understand the instructions that are entered - instead the entity who performs the programming (known as the *programmer*) will enter instructions in an intermediate language, which occurs at a *higher-level* than the machine. The machine itself only natively understands machine code; all other languages must be converted into this machine code, either directly through compilation, or indirectly through an interpreter.

'Programming' itself is an ambiguous term. It can be used to refer to the act of entering source code, the act of designing a program (either at a high-level or a low-level), the act of testing and debugging a program, or, more commonly, all of the above. For the purposes of this thesis, we see each act as subtasks of programming, which can be separated; for example, programming a single function does not require high-level design, apart from designing the purpose of the function as a whole. The systems discussed later in the chapter are concerned with varied areas of programming; however, we do not cover high-level design, as explained in Chapter 4.

This chapter first looks at Java, the programming language we have chosen to use as our tutoring language, and details the benefits over other languages. We then detail what programming is about, and the various constructs involved with programming. We then look at environments for programming, followed by environments for learning programming. Finally, we discuss what these systems have taught us about teaching programming.

3.1 The Java Programming Language

Java is an imperative, statically-typed, Object-Oriented (OO) language developed by Sun Microsystems, and originally released in 1995. It runs on a virtual machine (VM), by first compiling the Java language to Java VM bytecode.

Java has been hugely popular as an introductory programming language since its inception. It is a system-independent language (facilitated by VM implementations for many platforms), especially suited towards web programming; it has a mantra of "Write once, run anywhere", meaning that it should run the same way on any system that has the Java VM installed. As it is an OO language, it has also been used extensively in modern software design courses.

Java has many advantages over other languages for the teaching of programming. For one, its features and libraries are extensive, yet as Java code will need to exhibit the same behaviour on many platforms, many low-level operations are purposely abstract. These abstractions allow for a more stable and predictable behaviour, which means any errors in the program will be more understandable to the programmer; by this design, Java does not allow certain types of potentially illegal low-level operations (prevalent in languages like C or C++), reducing the chance of the student constructing obscure bugs. One of the common causes of bugs amongst novices in a language

like C are memory-allocation bugs (either not allocating enough memory or forgetting to deallocate memory). This can cause the system to behave unpredictably - the cause of program errors may not be immediately identifiable, or it may not even be apparent that the system is in a state of error. The Java language specification [Gosling et al., 2000] requires an automatic "Garbage-Collection" mechanism, precluding the need for manual memory allocation, and therefore preventing these kinds of errors.

There is also a feature of Java that makes it difficult to learn for novice programmers. A complete, working Java program requires what is known as "boilerplate" code; this is mandatory wrapper or template code that verbosely defines the shell of a program, but is essentially the same for each program. In the case of Java, this boilerplate consists of defining a class, with a "main" method with a particular signature defined inside, which contains the actual unique code to run the program. This repetitive code gives the impression of Java being not as "light" as some other languages (for example Python), where a single statement can be a complete program. Despite this, Java is still the language we have chosen to tutor for this project, as the other advantages outweigh this negative. Also, as you will see in Chapter 4, we approach the problem from the method level, not the class level, such that the boilerplate problem is essentially avoided.

3.2 Important Programming Constructs

Listed here are several programming concepts that programmers in an imperative language like Java should be familiar with once they have passed the novice stage. These are very important concepts for inclusion in an introductory programming curriculum.

- **Program**

  A program is formed, according to a specification (not necessary explicit) from a combination of the language constructs mentioned in this section. Not all are

required; in fact, the smallest program in some languages can consist of just a single statement.

- **Statement**

  A statement is the smallest standalone element that does not return a value. A statement can contain several subelements, including expressions. Some examples of statements include *Assignment* statements and *Return* statements.

- **Statement Block**

  A statement block, also known as a compound statement, consists of several statements and/or nested statement blocks. A statement block can be best thought of as a tree structure, with the statement block itself being the root of the tree, and the elements it contains being the nodes. Some examples include *If* statements and *Loops*.

- **Control Flow**

  Program Control Flow designates the order that the statements in a language are evaluated once the program is run. Control flow in an imperative language is sequential by default, moving from statement to following statement, but this order can be changed by control flow statements. Such control flow statements include loops and conditional statements.

- **Variable**

  Variables are elements which are named, and can store values. In Java, variables have a type as well, which specifies what the variable can hold. For example, some common variable types are ‘integer’ (can hold a non-floating-point number), boolean (can hold only a ‘true’ or a ‘false’ value), and string (text). Variables can have values explicitly assigned to them; in Java, this is done in an

assignment statement such as “myNum=88;”, which assigns the value 88 to the variable myNum. Also, a variable must be declared before its use; this is done by specifying the type then the variable name, as in “int myNum”.

- **Operator**

  These are special functions (special in the sense of how they are called compared to regular functions) that are often mathematical in nature, such as + (addition), - (subtraction), and * (multiplication). These often have the property of being called with infix notation (i.e. ‘4 + 5, 8 * 3’, as opposed to a regular function call notation, ‘add(4, 5);’).

- **Expression**

  Expressions are code fragments composed of atomic constructs that return values, and that can be evaluated as a whole to return a value. These atomic constructs can include operators, variables, and method calls (which point to composite structures, but are, in terms of their abstraction within the expression, atomic). An example of this is the right-hand side of the assignment statement “x = square(z) + 12;” - for this expression, the method call ‘square’, with variable ‘z’ as the argument, is evaluated, and the result added to the value ‘12’. This final result (i.e. the “value” of the expression) is assigned to the variable ‘x’.

- **Conditional/Branch**

  A branch, which is a type of control flow statement, allows a program to run or ignore a section of code, and optionally run or ignore one or more other sections of code, depending on a condition. The most common way of performing branching in Java is to use the `if` statement. An example is:

```
if(x==7){
```

```
    doSomething1();
} else {
    doSomething2();
}
```

In this example, when the `if` statement is reached, the program will proceed with execution in one of two ways; if the variable x is equal to 7, then the program will call doSomething1() but NOT doSomething2(), whereas if x is not equal to 7, then the reverse will occur. The 'else' clause is optional; if it were omitted in this example, then if x were equal to 7, doSomething1() would be called, and no code would be ignored, otherwise if x were not equal to 7, then the call to doSomething1() would not happen.

- **Iteration (Loops)**

  Loops allow the repetition of a section of code, usually repeating until a *loop condition* is satisfied. This condition depends on the purpose of the loop, and may include iterating over the indices or elements of an array or other sequence, therefore exiting when the end of the array is reached (i.e. performing an operation on all elements of a sequence); performing the same iteration, but exiting when a certain element is found (a "search" iteration); a 'main execution' loop, which will continue iteration indefinitely, performing tasks such a polling for user input and displaying information to the user, which only exits once the user has chosen to quit the program; iterating between two integer values or variables (i.e. 3 to 7), performing a running calculation on each value (an example being calculating a factorial); and many other possible applications.

- **Function/Method**

  Functions and methods, as well as the aforementioned branching, are important

in the paradigm of *structured programming*. Methods differ from functions, in that they are associated with objects; i.e. object X is the container of method Y, which when called, will have access to object X's properties. In Java, functions and methods also have return types which specify the type of value that must be returned from this method once the method has finished processing. Also, methods can have parameters, which allow each method to be specialised on a value passed, e.g. a method cookEgg with the parameter named 'time' could cook a (virtual) egg for however long the parameter 'time' is set to during a particular call to this function.

- **Scope**

  Scope can apply to a variety of components. With variables, it says where a variable can be seen from. For example if a variable is declared inside a block, then as soon as the program exits from that block, then the variable 'disappears', i.e. the program moves outside the variable's scope. A variable can be seen by anything following it at the same or deeper-nested level.

- **Object**

  Objects are the fundamental data structure at the core of OO programming. An object can contain fields and methods. Fields are properties of an object, i.e. variables, whereas methods are functions that are associated with the object.

- **Class**

  Classes are object blueprints. An object is just an 'instance' of a class. These classes contain descriptions of the fields and methods that the object will contain, as well as information about relationships to other classes (such as inheritance).

- **Recursion**

  Recursion is a technique where a function is defined in terms of itself, and allows

solving problems that consist of identical (in terms of solving them) subproblems; using recursion in this way is known as 'Divide and Conquer'. A recursive function must have one or more base (terminating) cases, which is a condition the function encounters where it does not need to recurse further; such functions without a base case will descend ad infinitum. Some recursive problems (but not all) can also be redefined in terms of iteration.

An example of recursion can be traversing a tree. Each call to the tree-traversing function falls under two conditions: a) the argument is a subtree, and the function recursively calls itself on all of the subtree's top node's children one by one, or b) (the terminating case), the argument is a leaf node, therefore the algorithm has nowhere else to go, and can return. This can be accomplished with a single function. All arguments are either subtrees or leaf nodes (the very first argument, i.e. the whole tree, can be considered a subtree).

3.3 Programming Environments

There are numerous programming environments available, both commercial and non-commercial, with various approaches. We can divide them into three categories: a) *Text Editors*, which are at their core concerned only with the editing of text, b) *Programming Editors*, which take text editors and augment them with helpful programming-related text-editing features, and c) *Integrated Development Environments* (IDEs), which contain a suite of tools (usually for program design and compilation), so that several programming related tasks can be accomplished within the same environment. IDEs are often distinguished by the coupling of design elements (for example, Unified Modeling Language (UML) diagrams) to code. Over time, the distinction between programming editors and IDEs has become blurred.

```
public class HelloWorld {
public static void main (String[] args){
boolean display = args[0];
if(display){
System.out.println("Hello World!");
}
}
public void aMethod (int x){
String aString;
```

```
public class HelloWorld {
    public static void main (String[] args){
        boolean display = args[0];

        if(display){
            System.out.println("Hello World!");
        }
    }

    public void aMethod (int x){
        String aString;
```

Figure 3.1: Java Source Code: plain vs syntax coloured and indented

3.3.1 Text Editors

Tools such as Microsoft's *Notepad* are environments that satisfy the bare minimum requirements for a programming editor; as source code is essentially plain text, a basic programming editor requires the ability to enter, edit, and persist this text. Although these tools can be used successfully to enter source code into files and also edit this code, several features that greatly aid writing code and understanding existing source listings (useful because of the potential complexity of programs) are missing from these tools; this is because these tools were not designed to specially cater to source code entering. These missing features are now ubiquitous in what people today would consider programming editors.

3.3.2 Programming Editors

Programming editors are specifically targeted towards aiding programming tasks, but still function essentially as a text editor. A programming editor has syntactic awareness of the source code to allow automatic code indentation and source colouring. Figure 3.1 shows a comparison between plain Java source code, and the results of the application of the following features, as displayed in the GNU Emacs editor.

- **Syntax Colouring**

  Syntax colouring involves colouring units of the source code (i.e. identifiers and code 'punctuation') differently according to their syntax. The colouring schemes are usually customisable. The right side of Figure 3.1 shows one possible colouring configuration for Java source code. With this particular scheme, scope keywords are coloured purple, class names and types are coloured green, method names are coloured blue, and variables in declarations and strings are yellow, whereas method calls and punctuation are black. Contrasting schemes may not just differ in the colour used, but also in how the source code units are divided amongst the colour groups (for example, we could instead distinguish between a parameter type and a declaration type).

- **Automatic Indentation**

  Indentation is primarily used to aid visual distinction of scope of a section of code. In our example, it can be easily seen that "`boolean display = args[0];`" and "`if(display){`" both occur as children of the method `main`, whereas "`System.out.println("Hello World!");`" can be observed as a child of "`if(display){`", and therefore a descendant of `main`. Without indentation, it requires more mental effort to deduce the exact structure.

  Although you can indent with simple text editors, this is manual and often error-prone due to the diligence required. With programming editors, automatic indentation occurs; when a change in scope occurs (for example, the line immediately after a method signature), then all following lines will be indented by one more 'level' (usually a single tab), until the scope ends (by a closing brace) or another level of scope is added (`if` statement etc).

In terms of these features, all programming editors are the same (except maybe for colour schemes and languages supported) - editors are distinguished from one another via more fundamental text-editing capabilities (for example method of input and key-bindings), therefore it is not important to talk in depth about a particular programming editor. Programming editors such as Emacs and VIM are the most popular programming editors, due to their extensive history and widespread inclusion in Unix and Unix-like systems.

3.3.3 IDEs

As full-fledged IDEs are essentially several development-related tools bundled together, they require more computing power, hence they have only become very popular recently as computing power has increased. Systems such as Eclipse and Visual Studio are the standard for software development IDEs, due to features such as Graphical User Interface (GUI) designers, and integrated debuggers.

Microsoft Visual Studio is the most well-known (and largest) IDE, due to it being the flagship IDE of the makers of the most prominent operating system, Microsoft Windows. To distinguish it from basic programming editors, Visual Studio has a large set of features, such as code completion, an integrated compiler and debugger, a GUI designer, and source-control integration. Other IDEs follow a similar integrated model to Visual Studio, usually with a subset of the features.

DrJava [Allen et al., 2002] is an educational development environment designed as an aid when teaching introductory Java and OO concepts, as well as allowing students to write complete software. DrJava contains an editor, a debugger, as well as a Read-Eval-Print Loop (REPL), similar to LISP, which allows the user to evaluate Java code fragments without compiling. A student can evaluate individual statements to see the

result, without needing to know how to write complete programs; being able to practice these atomic concepts early on provides a lower barrier to learning introductory Java, which is normally hindered by the amount of boilerplate code a programmer is required to write. DrJava also provides a language-level facility, which restricts the scope of Java a student can use in their programs, according to the level selected (Elementary, Intermediate, Advanced, or Full Java), each one building on the previous in terms of concepts allowed. For example, the 'Elementary' level allows `if` statements, but does not allow use of the `null` value, whereas the 'Intermediate' level allows `null`, but does not allow the use of loops. This aids a student in building up their knowledge slowly without being overwhelmed by the language features available.

A similar system to this is BlueJ [Van Haaster and Hagan, 2004]. The BlueJ interface is shown in Figure 3.2. BlueJ is similar to DrJava, in terms of the general interface (including the REPL), except that it adds to the environmental complexity with UML diagrams and its own graphical programming interface, whereas DrJava displays its programs using only code. It is argued that the graphical programming interface does not scale up, meaning users are more likely to 'outgrow' BlueJ [Allen et al., 2002]. Although neither of these systems contain learning content as such, they have been specifically tailored towards being used in a learning setting.

CAISE [Irwin et al., 2005] is a collaborative software engineering environment that employs a full semantic Java model, generated by a custom tool, JST (Java Symbol Table), and therefore has a semantic awareness of the source code. JST builds the model by identifying all the components in a Java parse tree, such as variables, methods, and classes, and then identifying all the relationships between these components. The model, which semantically describes a program in great detail, is used a) to allow general semantic awareness within the environment, for example accurately locating all calls to a given method, and b) to identify when collaborating users are working on

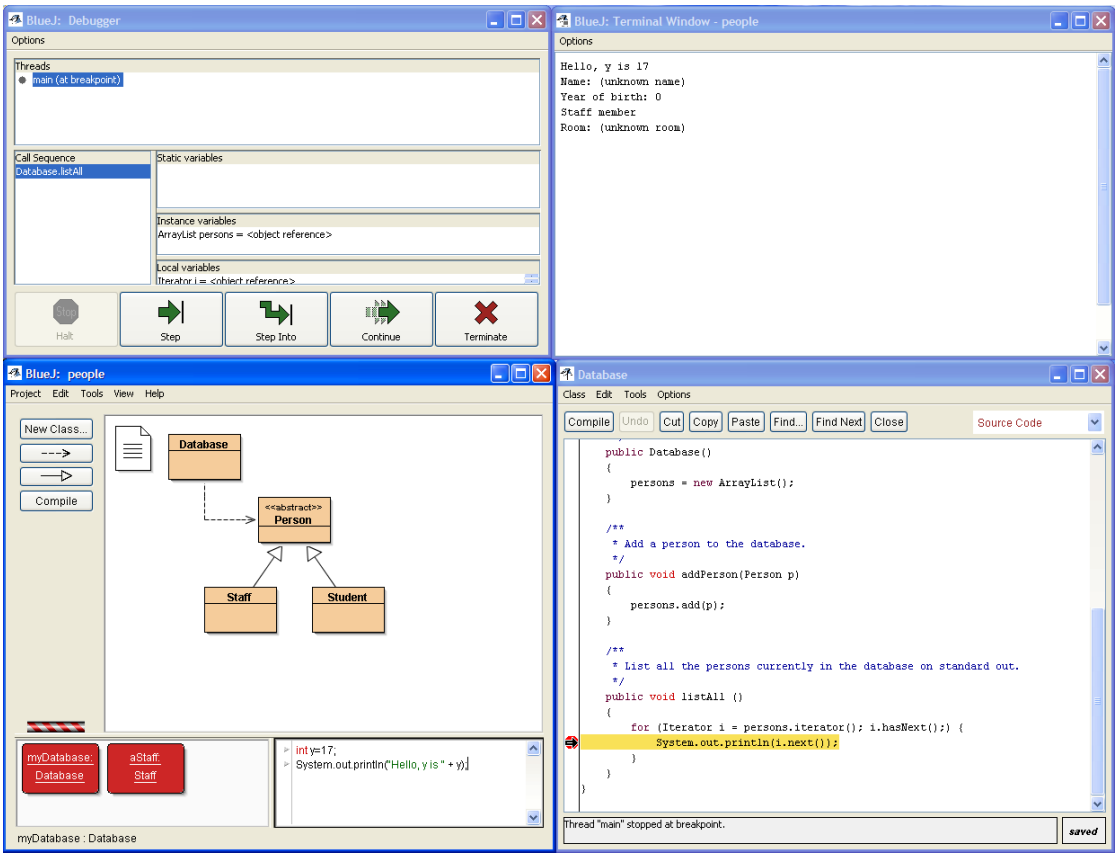

Figure 3.2: Interface of BlueJ

related components, and can inform users connected in this way when relevant components are updated.

3.4 Intelligent Programming Tutors

PROUST [Johnson and Soloway, 1984] is an ITS for teaching Pascal programming. Its core feature is the ability to analyse and understand buggy and correct programs, and to understand the program in detail; specifically, the system's goals are to identify what the bugs are, where they are, the student's true intention, and the misconceptions the student may be having. The system's "programming expert" component, which performs this analysis, is built upon the idea that program requirements can be decomposed (although not uniquely) into goals, which in turn are satisfied by the student through the implementation and combination of *programming plans*, which are procedures for realising intentions. An example of such a plan (as recognised by PROUST) is the *running total loop plan*, which is the procedure of computing a total by using a loop. The system attempts to recognise which plans the student is using, and interpret them as either correct or incorrect for the context, and whether there are bugs within the plans themselves. When there is ambiguity between which goal decomposition has been implemented (possibly because of an incomplete or buggy implementation), then PROUST uses a set of heuristics to decide which interpretation is more consistent with the rest of the student's implementation.

The performance of PROUST's programming expert was tested on 206 different novice solutions to a single programming problem. Of these solutions, PROUST was able to provide a complete analysis (i.e. process all of the code, without discarding any uninterpretable sections) for 142 of them (72%). Within these 142, there were 505 bugs in total, with 478 (95%) of them being correctly identified by PROUST, and the rest

were missed. In addition to these, there were 45 "false positives" (bugs identified that were not really bugs) reported. The authors stated it is not expected for PROUST to ever achieve a 100% overall correct rate, due to the existence of a) unusual and infrequent bugs that will never be in PROUST's knowledge base, b) novel plans that are hard to predict, and c) some ambiguous cases that can only be clarified by the student.

One of the most popular programming ITSs has been GREATERP (Goal-Restricted Environment for Tutoring and Educational Research on Programming), a LISP tutor developed at Carnegie-Mellon University [Anderson and Reiser, 1985]; it was also one of the first programming ITSs. A model-tracing tutor, it has provided a good starting point for other programming-tutor research. GREATERP contains a domain expert module that can generate complete LISP functions from problem specifications. The system gives the student experience in constructing whole programs, in an environment that resembles a real-world programming editor. As with other model-tracing tutors, the domain knowledge is represented using production rules; a example of a production rule for the domain of LISP programming, as implemented in GREATERP, is shown in Figure 3.3. The domain expert follows along with the student as they enter each symbol into their solution, validating each step as part of a correct rule, or a buggy rule; if the rule matched is buggy, then the system interrupts the student and gives advice, otherwise the student receives no feedback. The resulting interaction between the student and the tutor represents a dialogue. One kind of advice the system gives is the "planning mode", which is distinct from the main GREATERP interaction state, "coding mode". A student enters this mode when they are having difficulty. During "planning mode", the tutor explains through the algorithm for the problem step-by-step, and then returns the student to "coding mode" with the hope that the student now has the knowledge to continue.

An evaluation of the system showed that its effectiveness approached that of a human tutor; on average, students covered the entire course curriculum in 15 hours, which was only 3.6 hours worse than the average time taken for the students to complete the material with a human tutor (11.4), and 11.5 hours better than learning without either (26.5). Covering the material in a classroom setting takes over 40 hours.

```
IF    the goal is to combine LIST1 and LIST2 into a
      single list
THEN  use the function APPEND and set as subgoals to code
      LIST1 and LIST2
```

Figure 3.3: GREATERP production rule example

CITS [Zia et al., 1999] is an intelligent programming tutor for C++. For this system, the user modelling approach is different in the sense that a student model is built by focusing on domain-independent attributes (such as verbal, abstract, and numerical reasoning) rather than on domain-specific skills. Also, due to this approach, and because student models can sometimes require a lot of information before they stabilise (i.e. before the student model becomes an accurate representation of the student's knowledge), the authors have implemented stereotypes, which contain information about attributes that often co-occur in groups of people, to make predictions of the actual model of the student before sufficient user-model completion information has been acquired. An aptitude test is given before a student uses the system to determine which stereotype the student falls into; this stereotype is used to initialise the student model. The lessons themselves are also categorised using the reasoning metrics - this is achieved by dividing the C++ concepts into the three categories, and giving each lesson a value based on what concepts they contain. For example, one verbal concept is functions, and one numerical concept is operator-overloading.

The stereotype is also used to choose the amount of media to present the lessons to the student. The articulatory distance (the distance between the meaning of an expression in the interface, and its form) for each problem is calculated before each lesson; the higher the distance, the more media is required. For example, if the lesson is textual, and the student has low verbal reasoning, then the articulatory distance is high.

The Java Intelligent Tutoring System (JITS) [Sykes and Franek, 2003] is a Java tutor that allows free-form coding, albeit with no design section. The curriculum is a small subset of the Java language (variables, operators, and looping structures). The system is built around the core of the "intent-recognition algorithm" [Sykes and Franek, 2004]. Several strategies are implemented to attempt to predict what the student was intending to accomplish with his code. One such strategy, the "Syntax Error Correction Strategy", comes into play if the Java parser does not succeed. This strategy finds unrecognisable tokens in a students submission, and reverses possible error transformations such as the replacement of a symbol by another symbol, or the insertion of an extraneous symbol, to attempt to correct to code; if a more meaningful code chunk is obtained, the system assumes that a syntax error is present, and that the corrected code chunk represents the true intent of the student. JITS will then initiate a dialogue with the student, with feedback being in the form of questioning sections of the code; for example, the system may ask "I see 'intt'. Do you mean the keyword 'int'?". The dialogue continues until the syntax of the code is completely correct, and can be fully parsed.

For semantic errors, JITS breaks down the requirements of the problem into a high-level functional decomposition tree, to divide the problem into discrete sections. From this tree, a decision tree is generated that is used to validate the solution, and also to provide context for feedback (each node of the decision tree is associated with feedback if the concept of the node does not pass). For example, for a problem that requires a `for` loop, one of the nodes may require the `for` keyword to be present; if it is not,

feedback is given to the student, otherwise the system traverses the next node in the tree, which checks for the appropriateness of one of the loop variables. This decision tree is only used once the solution can be parsed without syntax errors. The user interface is presented like a simple editor, but with a problem statement and an output window to allow the student to run their solution and see the output.

Kumar's set of C++ and Java tutors [Kumar, 2004] each focus on a single skill and tailor the interface to that particular skill. They generate three types of problems; debugging problems, output prediction, and expression evaluation. Amongst other topics, they teach encapsulation [Kostadinov and Kumar, 2003], scope [Fernandes and Kumar, 2004], `for` loops [Kumar and Dancik, 2003], and C++ pointers. The tutors use randomly generated problems to prevent plagiarism (i.e. students copying other students' answers to the problems). Also, the generation is adaptive, such that it will only generate problems for topics that the student has not mastered. As an example of a tailored interface, the Parameter Passing tutor [Shah and Kumar, 2002] contains an interface component which quizzes the student on the values of variables pre- and post-method calls. This component lists the values that these variables have been initialised to.

BITS (Bayesian Intelligent Tutoring System) [Butz et al., 2004] is a web-based ITS for teaching Java programming. Despite the name, it is debatable whether it is an intelligent tutoring system; a more apt description would be an adaptive courseware system. Using a curriculum mapped to a directed acyclic graph (in the sense of "knowing concept *a* relies on knowing concepts *b, c,* and *d*") within a Bayesian network, the system stores how much the student knows about a concept, and the probability that they know other concepts, so that the system can guide the student through the material in an appropriate way; the material covered includes elementary programming knowledge, such as variables, assignments, and control structures. The system creates this student-model by asking the student whether they understand the topic or not; if the student is unsure,

then they can request a short multichoice quiz on the concept. There is no real problem-solving, therefore debatable educational value; the student is not required to perform any coding or program design.

RoboProf [Daly and Horgan, 2004] is a Java programming tutor. RoboProf evaluates the solutions by compiling and running the Java programs, and examining the output. For each submission, the system is run against test data, and displays to the student a) this test data, b) the output that the student's solution actually created, and c) the output that was expected by the system (i.e. the 'correct' output). Evaluation entails simply comparing the output values between the actual and expected. An example problem given by the authors showed four sets of test data, i.e. four calls of a method using different data, to produce different output. To prevent students cheating (i.e. submitting once to see the test data, then hard-coding the expected output values), the system can randomly generate the test data. This is a shallow method of evaluation though; examining a limited set of output values does not guarantee that a program is truly valid for the requirements - it may fail on extreme cases, for instance. Also, by not looking at the code itself, the solution may be valid, but it may not be *good*; it could be inefficient, either in terms of performance and/or space (using several elements to perform a function that has already been captured by another existing language construct). Also, it is unable to give proper feedback about the solution itself; all the system can do is inform the student that it was incorrect, and show the expected output. As RoboProf does not provide accurate and fine-grained evaluation, and does not keep a student-model to adapt instruction, it cannot be considered an intelligent tutoring system.

CIMEL ITS (Collaborative Constructive Inquiry-based Multimedia E-Learning) [Wei et al., 2005] is an ITS for teaching Java, with a design-first philosophy. It uses a variety of methods, including multimedia instruction. The instruction is split into two sections; during the first section, the student is taught the concepts and given

simple quizzes based on this material, and in the second section the student performs actual coding, solving problems in the Eclipse IDE. These 'coding' problems are very specific about what is required, which reduces the range of possible solutions and would simplify evaluation. For each 'coding' exercise, the student is first given a set of classes, then they are expected to write code to create objects from these classes. It is not explained how the expert evaluator would semantically validate the solution; this is unfortunate, as static semantic reasoning about solutions is difficult, especially if the system could expect a wide range of solutions from students, and it would be interesting to know their approach. The system also keeps a 3-layered student model, with the different layers recording how well the student understands relevant concepts (Problem-domain Knowledge Model), the historical knowledge state of the student (Knowledge Model), and the student's general problem-solving patterns and anti-patterns (Cognitive Model).

3.5 Discussion

Programming environments can be multi-functional and powerful, but they overwhelm the user, and do not teach programming itself. Some give help with the available commands and library functions, but they don't explain what programming is about - the language-independent programming concepts are not apparent in the materials the environments give. Instead, this help serves as more of a reference. They give help on *languages* and functionality extension, not on programming itself. This distinction is important. Also, you can program, but can you program well? Programming environments do not serve to restrict what the user can do in order to make the code better. This is a feasibility issue as well as a freedom issue; with no formal way of specifying to the environment the intent of the project (i.e. the "problem" the developed software is

supposed to solve), the environment cannot understand what code is appropriate or inappropriate for the project - there is no semantic validation. Software design languages like UML allow the user to specify certain purposes of the program (through use-case diagrams), but it cannot describe in such detail or in an unambiguous way that can be programmatically mapped to code. It would be possible to generate such a language, but the feasibility of unambiguously describing in detail a project, which description can then be used to validate the appropriateness of code for the project, decreases very quickly as projects grow larger. Even the programming environments that are built for educational purposes, such as DrJava and BlueJ, only provide an environment to make it easier to understand language concepts for novice programmers (if they are working on their own projects), but do not provide instruction and semantic validation themselves.

Programming ITSs, and the methods of tutoring, are varied; this is apt for such a domain as complex as computer programming. Also, because of this, there are numerous levels you can tutor on, and numerous concepts in each level. Tutors such as PROUST, JITS, RoboProf and GREATERP give the student experience with writing complete programs, taking a real-world approach to the interface, whereas other tutors, like Kumar's encapsulation tutor, focus the interface on tutoring a particular skill. For the tutors that allow the student to write complete programs, the semantic validation is an interesting area of discussion; semantic validation of programs is, by the very nature of the domain, hard. Solutions to programming problems, except in trivial cases, are not unique: values can be changed, different operators can be used, and even structural changes can be made, and the solution will still be valid and equivalent in terms of satisfying the problem requirements. Accepting all the correct solutions (and rejecting all incorrect solutions) is a difficult task, and not one that offers a single obvious answer.

GREATERP solves this problem by not allowing true free-form coding - as with other model-tracing tutors, the path to creating the solution is restricted. A student can-

not create an entire solution and then ask the system to evaluate it - feedback is only available at each step in the creation of the solution. This method reduces the difficulty in evaluation (the system essentially encodes all the explicit possible solutions in its domain model), but it also reduces the freedom of the student, and reduces the total space of valid solutions (if the domain model is incomplete, either intentionally to reduce design time or unintentionally through overlooked concepts). The student may gain misconceptions about domain concepts if a solution that they decide to implement can not be created due to the ITS's restrictions. RoboProf allows a greater range of solutions (in fact, all correct solutions should be accepted by the system), but has two major downsides; it does not give fine-grained feedback (which is important for supporting learning, especially in a domain such as programming with many concepts), and, as it only analyses the output and not the program itself, may accept incorrect solutions (as checking output on a few input cases does not guarantee correctness for all input cases - only by validating the logical structure of the program can this be done). JITS and PROUST's approaches show the most promise (in terms of accuracy of validation and freedom of solution form); there is explicit mapping of the problem requirements to a high-level description of the expected program. PROUST decomposes the goals of the problem into programming plans, and then matches these plans to part of the input. This idea is intuitive and potentially accurate; what determines the accuracy of the validation is the performance of the system in the act of matching the plans to parts of the solution, and correctness of the design of the plans themselves. JITS employs a similar method; it creates a functional decomposition, rather than a decomposition based on programming plans.

The curriculum for novice programmers has a common base amongst the systems: there are a set of concepts that are innate in programming itself, such as variables, conditionals, and iteration. A deviation in this curriculum has occurred since the intro-

duction and widespread adoption of OO programming. It is a belief amongst some that tutors for introductory programming that teach OO design, before or simultaneously with coding, can overwhelm the student with two sets of concepts that require vastly different cognitive skills. This is the view we have taken in the design of our system. This position can be further justified with the core idea that OO programming is an extension of procedural programming; object-orientation is used to structure programs at a higher level, rather than supplant the lower-level procedural concepts. Therefore, teaching OO first followed by procedural programming makes less sense than teaching procedural first, as with the latter approach, you are building upon what they have previously learnt. Teaching procedural programming first therefore is less confusing for the student; OO programming can be explained to the student as procedural programming, but with the addition of abstract data types (provided by the class and object concepts) that can contain both fields and methods, which gives the student a more structured way of organising larger programs. When teaching OO programming, there still exists a procedural nature to the programs that students write - the flow of control inside methods is still procedural, so they are having to learn two things at once. When procedural is taught, students only need to learn one method of program flow - the beginning of a program to its end can be simply traced. Once this concept has been learnt by the student, it can be built upon with OO theory if required. Also, the OO methodology only proves its worth once the procedural concepts have been learnt anyway; OO programming is useful for higher-level program design, whereas for small algorithms (i.e. containing few methods), it adds unnecessary complexity.

CHAPTER 4

Design

The design of a tutoring system involves many stages. Firstly, our approach to solving the 'Tutoring Programming' problem has to be decided upon. This decision influences what happens within the rest of the design steps. A curriculum (i.e. what skills the system teaches and in what order) also has to be carefully devised; possible influences upon this area include the skill-level of the intended audience of the system, as well as the curriculum of courses that are readily available in which to trial the effectiveness of the system. The nature and format of the problems are then designed; once we have the format we can then generate a problem set that sufficiently covers our chosen curriculum. Domain-model design is the next step - as the chosen representation is constraints (in the context of CBM), this involves analysing the domain to determine the syntactic knowledge units of the domain, and translating these units to natural-language syntactic constraints. Finally, our method of verifying semantic correctness has to be decided; this is difficult for such a complex domain as programming, and therefore requires thought. This involves the combined analysis of the domain and problem-solutions, as well as the design of semantic constraints. This chapter presents all the design decisions involved with developing the system.

4.1 Approach

A goal of the thesis is to increase the effectiveness of tutoring programming to students. Programming itself is a complex task and involves many different types of skills, often being utilised simultaneously; these skills include program design (at various levels), abstraction, and writing syntactically correct code. The student may fail to learn these skills properly if, while performing a programming exercise, the skills or the properties of the skills are not brought to the student's attention, and therefore is unaware that these skills are required. Without knowledge of the complete programming skillset, and therefore no way of 'managing' a complex programming task, a student will have difficulty understanding the nature of programming, and be overwhelmed by the complexity of the task.

There are several ways of making the student aware of these skills; for example, you may create an interface that makes the skill explicit to the student, but this approach comes at the expense of tutoring how to combine this with other programming skills. This may mean that the student understands a particular skill, but is unaware of how to utilise it to perform the actual act of programming. Specialising the interface in such a way that the behaviour is fundamentally different from a typical environment may result in transfer problems when the student moves from the tutoring system to a real-world development environment. An example of such an interface exists in the Parameter Passing tutor [Shah and Kumar, 2002], discussed in Section 3.4, where current values of variables are listed pre- and post-method calls. Though this is a useful addition, the core interface does not permit code creation like a programming interface would, and therefore gives no experience in program writing.

Although this approach is a valid approach in teaching the particular skill, there is the possibility of transfer issues that we noted before, as well as the fact that another

system will be needed to learn programming itself. We want to focus the system on helping the student solve problems and write programs rather than to focus on one particular skill; therefore the student will need exposure to several programming skills, and also how to combine these skills in program creation. To tackle these issues, the overall direction of the system is to provide a true-to-life programming experience, which will both help reduce transfer problems and provide experience with a wide range of skills, including syntactic and semantic understanding of programs, program control flow, and simple software construction. A major decision as part of this philosophy is that the interface design goal here is to create an interface that closely resembles a typical programming environment; the interface will not be exactly the same as a typical programming interface, but any major differences will be *augmentations*, and hence the core interface will still resemble a programming interface. If students were able to learn using such a system, then once they do graduate to a real-life interface, the required mental translation (transfer) would be small.

These augmentations provide a unique approach to the tutoring programming problem. The J-LATTE system aims to help students understand programming, by tutoring students in the skills needed to manage program complexity in basic programming tasks (defined here as tasks which do not require architectural design). The complexity we are interested in is when students must deal with writing syntactically correct code, whilst also having to think about the higher-level concepts. Our approach is to lessen the cognitive load, and also teach simple skill-separation by encouraging (but not forcing) the student to focus on one of these tasks at a time, by dividing the solution task explicitly into *two modes*.

The two modes are made possible through the system handling the *concept abstraction*. Students will design programs initially with certain abstract concepts from Java, without worrying about the code level complexity, in what we call *concept mode*. Pro-

gramming at the code level is still very important for learning programming, therefore the student is still expected to enter code at some point, by entering *coding mode*. This will occur after the student has created a complete or partial solution outline using the abstracted concepts; at the very least the student must have one concept in the solution before proceeding to coding mode. The student can work at the code level whenever they feel comfortable. The two modes this approach implies are working at a concept level, and working at a code level.

The level we chose for the concept abstraction was at the statement and block level. Each abstracted concept either represents a whole Java statement (i.e. 'Assignment', 'Declaration'), or a Java block, containing many statements (i.e. 'For Loop', 'If Statement'). This was chosen as statements and blocks provide naturally-occurring divisions in Java (and in several modern programming languages). The concepts selected for abstraction map directly to Java statements, which means it is obvious where each concept fits into code, creating less translation for a student working with these abstractions. These particular concepts were chosen because we looked at all possible statement- and block-level actions, and chose ones which were relevant for our problems. The chosen concepts are 'Declaration', 'Assignment', 'Print Statement', 'Return Statement', 'If Statement' and 'For Loop'.

Other representations are possible. The main candidate, apart from our chosen representation, was the potentially language-independent form of pseudo-code. Pseudo-code is an ad-hoc way of describing a computer program or algorithm that is in a form that is closer to human language, often (but not necessarily) in sentence form. An example of pseudo-code is shown in Figure 4.1. Pseudo-code is useful because for novice programmers it may be easier to understand code if the grammar and the vocabulary is familiar, and express code if the grammar is less restrictive than a computer language.

Increment variable x *by 2, then multiply it by 5 and return the value.*

Figure 4.1: Example of an algorithm represented in pseudo-code

Also, with a comprehensive pseudo-code representation, it would be possible to design a whole program first without entering any code in a computer language.

This was rejected as there is no standard way of writing pseudo-code, therefore we would have to devise our own representation. Human language is complex and ambiguous, and real-life pseudo-code inherits this ambiguity; like human language, the language used to form pseudo-code can have more than one word for the same concept, and a programmer may choose to use a higher-level abstraction rather than express the details of an algorithm, if they are familiar with a concept. An example of varying terms can be demonstrated with our previous example in Figure 4.1; instead of "increment variable x by 2", an equivalent statement would be "add 2 to variable x".

Because of these reasons, it would difficult to devise a representation that was, for all required cases, comprehensive, consistent, and unambiguous. Also, despite the resemblance to natural human language, a student would still need to be familiar with programming terminology and concepts, such as variables and return values.

There is also the issue of validating solutions created with pseudo-code against the problem requirements. This involves translation of the pseudo-code into some Java representation. Our chosen form is very close to Java so therefore avoids this problem. Following from this, we would have several possible choices for the specificity of the pseudo-code; we could make the pseudo-code very language-independent, which would be more difficult to translate into Java and validate but good for teaching general programming concepts, or more Java-specific, which would be easier to validate and design but the general programming concepts would not be so obvious.

Also, we want to encourage people to think in a code-sense; forcing the student to think in pseudo-code may slow down the process of forming the solution once they

become more proficient; one option would be to give the student the choice to switch the representation into a more efficient form, but this flexibility is beyond the scope of our research, as we would need all our solution diagnosis code to work for both representations. So, although a pseudo-code approach would be very appealing in terms of providing less of a barrier to a novice programmer, it was not feasible.

For the concept-abstraction approach that we decided on, initially the decision was to enforce a 'high-level first' philosophy by only permitting one-way movement between concept-mode and coding-mode; it would be mandatory to have the complete and correct concept outline before moving on to coding, the result of this being that students would be forced to think at a more abstract level before rushing in. This approach was implemented initially but was rejected for two reasons; firstly, it was noted that once problems became more complex, the concept representation did not provide enough detail for the system to diagnose the solution. Secondly, after some initial testing, it was found that the concept representation was too sparse for the student to remember the intended purpose of each concept when dealing with a complex solution with several concepts. By coding after placing only a few concepts, the student is able to use this code detail to act as a reminder of the purpose. They are able to leave a section of the solution and work on another part, then come back and remember what the intention was due to this detail. Due to these issues, we allow the student to move freely between the two modes.

A good pseudo-code representation would not have had these problems, due to the detail that we could have specified within a pseudo-code statement, but, as mentioned before, such a representation was unfeasible within the scope of this project. Our chosen representation gave different benefits, and was believed to be sufficient for the project goal. By discouraging entering code first, the student is encouraged to think about the higher level first. As noted before, the separation is not rigid; the student can freely

move between the two modes. This is because concept abstraction does not give much information; with larger, more complex programs, code may be required to be implemented to clarify the purpose of a subsection of the program, or if the student designs it a certain way, they may create a general design, then complete each subsection with code and more concepts before completing the rest.

Another advantage of the concept mode/coding mode method is that the code the student writes is explicitly associated with a concept. This removes ambiguity from the student's solution if the student has entered incorrect code for a concept - in some cases the system may not have been able to identify the true intent of the student if the code was without concept context. This in turn allows the system to be more precise (and hence useful) in its feedback.

4.2 Curriculum

Java is a very substantial language. Although a tutor which addresses all concepts in Java is ideal, for Masters research this sort of breadth is unfeasible. Only a small percentage of concepts could make it into the J-LATTE system, so only the most crucial topics were chosen for inclusion. All possible concepts were identified, and the precedence for each one was decided for inclusion in the tutor.

The most important factor in devising the curriculum is the intended audience of the system. As the ITS will just be reinforcing concepts, rather than explicitly teaching new ones, the students need to have at least touched on the material in lectures. The decision was to aim the system at novice programmers, who have been learning Java for approximately 6 weeks of a tertiary-level Java course. The reason this level was chosen was that simple concepts would be good to trial a new system around, but it would be hard to build a large and varied problem set around them; both the problems and the

solutions required would have to be simple. At the 6-week mark at the university that the system was to be trialled, the students would have just been taught loops. A control structure such as a `for` loop enables iterative program flow, allowing for more complex problem requirements.

A comprehensive list of possible curriculum elements was composed, and the priorities of these elements were calculated through various research activities such as exposure to lectures, talking to human tutors and students in labs and observing problems that students encountered. Much of the precedence was decided by approaching the system in a 'building block' fashion; as our audience is the novice programmer, we identified the most atomic concepts, and determined other components in which the original atomic concept is treated as a direct part, and gave these higher precedence. Important concepts with high priority include General Syntax (e.g. Braces, Parentheses, Blocks), Variables (Declarations and Assignments), Conditionals (If and Switch statements), Loops, Methods (definitions, invocation), amongst others. Certain concepts were also identified as being too complex or too obscure to include at this point; Events, Package definition and the 'super' keyword are just some of these.

The raw curriculum is then used as the foundation for designing the problem set and constraints. The design of the system is such that the curriculum can easily be expanded in the future, with more problems and constraints being generated. The final topics chosen for the curriculum were General Syntax, Variables, Strings, Methods, If Statements, and Loops.

4.3 Problem Design

In terms of the problem set, with many ITSs the goal of each problem within the system will be similar, with a general template being used to generate further problems in the

same goal set. No matter how complex the problem becomes to solve, the semantics that the constraints need to check are always based on a single general outcome. With programming, the area is so broad that different skills are required by programming solutions to problems with vastly different outcomes that are hard to generalise. Therefore, in a programming tutor with a free-form design such as J-LATTE, it is difficult to keep to just one form without restricting learning. To solve this issue in J-LATTE and to help guide problem development, we gave the main focus to an aspect within each problem; the end result is that we can say each problem belongs to a certain problem 'style', with an ideal solution (explained later) and several semantic constraints being generated around that style. Within these styles, the problems could be modified slightly, for example adding extra clauses, to easily generate more problems without generation of new semantic constraints. The three types of problems we developed were simple expression printing, predicate method, and simple iteration.

A 'simple expression printing' problem requires a student to complete a method such that it prints out a value. This value may or may not be the result of some calculations that the student is required to perform. An example of this type of problem is:

```
"Complete this method such that, when run,
 it will display the arguments multiplied together."
```

A 'predicate method' problem requires a student to complete a method such that it returns a Boolean value (true or false) based on a set of conditions. Each condition is stated separately. An example of this type of problem is:

```
"Complete this method such that it returns true if the length
 of the given name is no greater than 20 characters long. Use
 an 'If' statement to carry out the comparison."
```

A ‘simple iteration’ problem requires a student to complete a method such that it returns a value that must be calculated using a loop or set of nested loops. An example of this type of problem is:

```
"Complete this method (using a 'for' loop) such that it a)
 adds the squares of all the integers from the
 first parameter (startNum) to the second parameter (endNum),
 including those numbers, and b) returns the result.
 Assume that startNum is less than endNum."
```

Each problem is presented with its own *context*. The context is a code fragment that frames a problem, so that the student has existing properties to work with. For example, the context could be a `for` loop beginning and end, or a method outline (signature and braces). Often there will be variables and other methods that the student can reference in their own code, such as arguments to methods; in fact, often these variables will be mentioned directly in the problem text itself, and therefore the student will be expected to use them in some way. Also, the student is only given as much as is needed for the problem; it is unnecessary at a novice level to confuse them with Java class definitions and `main` methods, when all that is required from the student is the completion of a method independent of those constructs. An example context is as follows:

```
public void doSomething(int x){

   /* input area begins */


   /* input area ends */
```

```
}
```

(the /* */ comments indicate in this example which area the student is allowed to enter a solution in (i.e. non-“context”))

4.4 Constraint Design

J-LATTE uses constraints to model the domain. Constraint acquisition can be a time-consuming process, especially with a large and varied domain such as programming. This process is made easier and more structured through the method of constraint acquisition via an ontology. The Constraint Acquisition System (CAS) [Suraweera et al., 2005] provides an ontology-driven method with which to engineer constraints. Once a suitable ontology has been composed, example problems can then be added to the system, and from there we can generate constraints. The constraints produced are written in natural language, which can then be manually converted to the target language. Through CAS, an ontology, illustrated in Figure 4.2, was developed to represent the Java domain, through listing the main domain concepts and arranging them into a hierarchy. This hierarchy represented the “is-a” relationships within Java - for example, all concepts in the ontology are of type “Element”, a “Variable Declaration” is a type of “Declaration”, a “Parenthesis” is an “Enclosure Element”, and so forth. Non-hierarchal relationships, such as containment, were also specified between components. A complete Java ontology was not developed due to the scope of the research; the goal was to keep within the curriculum.

The tool was not able to generate a complete set of constraints in the time given, due to CAS being a work-in-progress at the time we were designing the system, the vastness of the Java domain, and the author’s inexperience with ontologies; the output of the tool

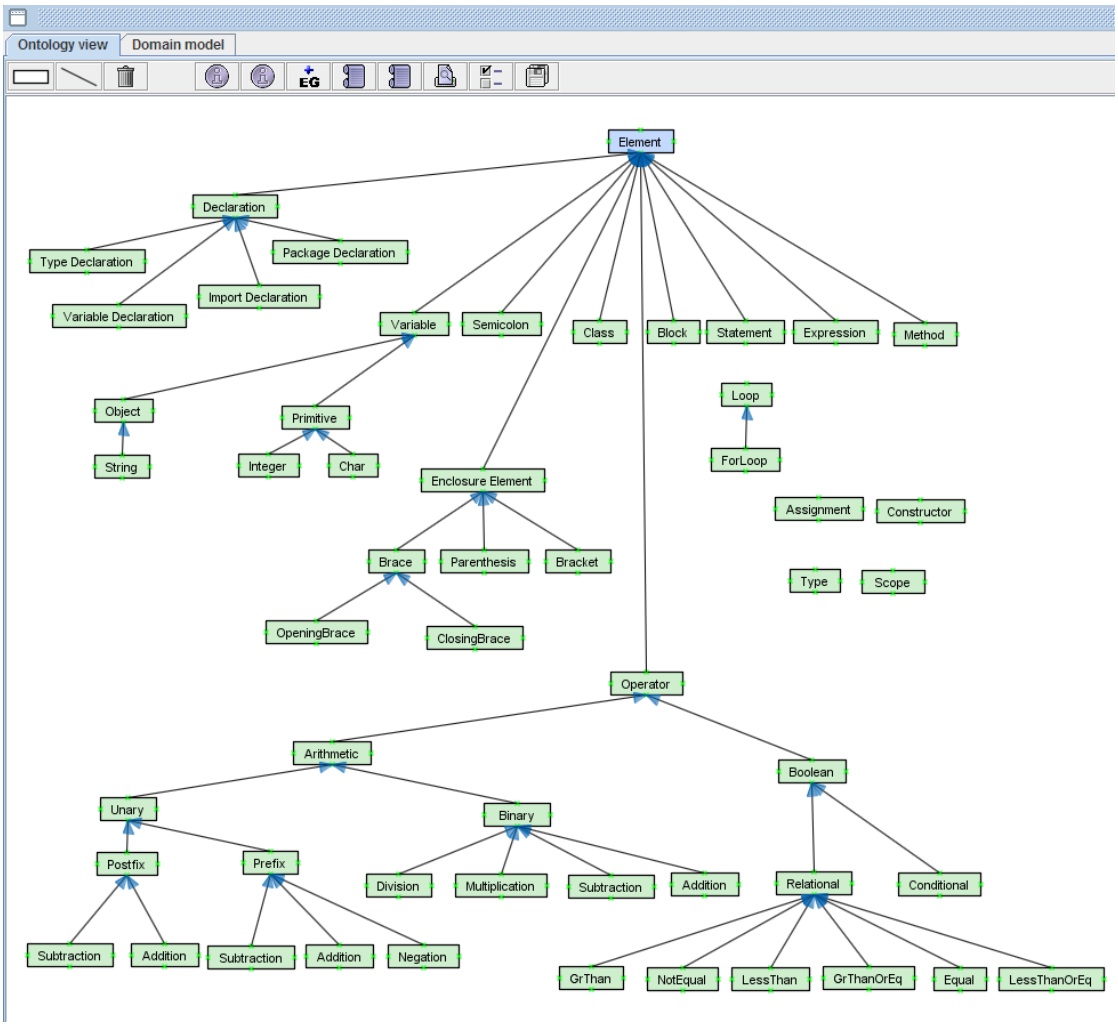

Figure 4.2: Java domain ontology, as developed in CAS

was only used as a starting point. Manual constraint development followed this, and was used for most of the development.

Three types of constraints were developed for J-LATTE: syntactic constraints, semantic constraints, and style constraints. Semantic constraints are discussed in Section 4.5.2, as their development is better explained in the context of verifying semantic correctness.

4.4.1 Syntactic Constraints

Syntactic constraints are easy to design, as they can be derived from the Java grammar. However, the number of constraints required for complete coverage would be too large due to the vastness of the Java domain, therefore only grammar rules that were relevant for the concepts selected for concept abstraction were considered. Figure 4.3 is an example of such a constraint. In this example, the grammar states that the right side of an assignment operator (in the correct context of being on the right side of an expression) must be a valid expression. The figure shows this grammar, as well as the natural language constraint derived from it.

Grammar:

```
Expression:
Expression1 [AssignmentOperator Expression1]
```

Constraint:

```
IF    there is an assignment operator
THEN  it must be the case that there is a valid
      expression on the right-hand side of the operator
```

Figure 4.3: Syntactic constraint design: from grammar to natural-language constraint

4.4.2 Style Constraints

A goal of the system is not only to teach programming, but also to teach good programming style. A student may code a correct solution, but the solution may be an inefficient way to satisfy the programs requirements. Examples of such issues in program code include overcomplexity (i.e. when a solution uses a set of constructs to perform an operation where another single construct, which performs all of the tasks in one call, would be sufficient), or performance inefficiency (i.e. when a solution uses an algorithm or set of constructs that performs operations that can be optimised). A small set

of ‘style’ constraints have been developed to encourage good coding style; a student will learn the best ways to perform an operation. As style is such a broad concept, and can apply to different components and sets of components in different ways, these constraints operate at different levels, sometimes over structure and sometimes over code itself. In accordance with the philosophy that the system does not unreasonably restrict the possible solution space, they are only used when necessary, and are designed to be ‘loose’. Some of these style constraints were common-sense, whereas others were designed through discussing possible correct solutions for problems with Java educators, and noting their preferences. An example of good style that is enforced by our system is that of return values from a method; it is considered good style by some to return a value once, and only from the end of method, rather than several occurrences throughout a method.

4.5 Verifying Semantic Correctness

Computer programs are complex and flexible entities. In any major programming language there are enough atomic components of varying functionality to make coding an algorithm a non-deterministic affair. What this means is that there may be several possible ‘correct’ solutions for a given programming problem; some may appear quite similar, whereas others may vary wildly in their structure. An example of this is shown in Figure 4.4. The figure shows a fragment of a solution; this fragment is a logical condition that should only return true if a variable’s length is less than or equal to 20. This fragment can be written in two ways to be logically equivalent. Other variations are possible that would still be valid; some of these have structural changes. To verify if any solution, even a large and complex one, is syntactically correct is trivial; compilers perform this task successfully, albeit with less useful feedback than our approach due to

a) having to operate on unpartitioned source code (our code is partitioned by concept abstractions), and b) a compiler's messages not being designed for instructional purposes. The real difficulty with programs is with verifying semantic correctness in regards to a problem.

"Complete this method such that it returns true if the length of the given name is no greater than 20 characters long. Use an 'If' statement to carry out the comparison."

Solution 1:

```
if (name.length()<=20){
```

Solution 2:

```
if (!(name.length()>20)){
```

Figure 4.4: Example of multiple valid solutions to a single problem

The approach to verifying semantic correctness within existing CBM tutors is to pair each problem with (in keeping consistent with the terminology of other constraint-based tutors) an *ideal solution*, and have semantic constraints perform evaluations using a given problem's ideal solution. We decided that an ideal solution, for this domain, should contain a representation of what is required from the student, but not down to a level of detail that refers to one particular solution; to accurately capture the range of valid solutions for such a complex domain, the solution must represent a solution at a higher level than what the student submits. Many possible correct solutions can 'fit' into the ideal solution; therefore, our ideal solution can be thought of as a schema or outline. An ideal solution may state that the student solution requires a list of properties, and it is up to the student how these requirements are met, using the language and components of the domain. The semantic constraints 'compare' the student's solution to the ideal

solution, across a varying granularity of domain knowledge, and report back on the correctness.

This method of verifying semantic correctness is present in the J-LATTE system. Each problem has a corresponding ideal solution, but this solution is stated in a different language than the student solutions, due to the preference that an ideal solution is at a higher level than the student's solution. The difficulty for programming, as compared to other domains, is that there can be greater variability in student solutions, especially with larger programs. In order to produce a system which can handle this variability during solution diagnosis, more care must be taken when developing the ideal solution language. This language is discussed in Section 4.5.1. Within J-LATTE, semantic constraints have been developed that refer to both the student and ideal solution in order to perform solution diagnosis in a semantic sense. The actual development and details of these constraints is discussed in Section 4.5.2. The solution is not compiled and run, and therefore no output can be checked; a solution is validated purely by evaluating constraints, and therefore is using static analysis, which we believe can potentially diagnose a solution more accurately than purely monitoring program output.

To overcome some of the difficulty of statically analysing solution code to verify semantic correctness, the problem statements were sometimes more explicit and therefore restrictive in their requirements than would be preferred. An example is the statement "Use an 'If' statement to carry out the comparison" from the 'predicate method' problems; in order to give the student some experience with 'If' Statements, which they might skip if they could place the boolean values directly in 'Return' statements, and also because of time restrictions, it was decided to restrict this by forbidding it in the problem statement. Future revisions would preferably support this option in a student's solution.

Other possible candidates for semantic evaluation include symbolic execution [King, 1976] and the use of formal methods. Symbolic execution works by presenting the inputs symbolically - these represent classes of inputs, rather than specific values. The result of symbolic execution will be a symbolic expression that can be examined to see whether the given input class will give the desired output across all cases. While useful for testing the overall correctness of a program and for testing correctness over classes of inputs, it does not give feedback on the program at a fine-enough grain to be a complete replacement for semantic validation for an ITS. Still, this approach may be useful as an addition to the system, to further confirm a program's correctness.

Formal methods would be useful, as they would allow us to give the program required for a problem a mathematical definition. These definitions would take time to develop and prove, and therefore were not implemented for this project, but may be incorporated in a future version of the system.

4.5.1 Ideal Solutions

The ideal solution describes a high-level version of what is required from the submission; rather than explicitly specifying what design concepts and code fragments should occur in the submitted solution, it only notes the general requirements of the given problem that *must* manifest in the solution for the problem's tasks to be considered satisfied, such as "the method argument length must be less than 10" or "must loop up to this variable". It is essentially a list of requirements, or a formal specification of the problem statement. As further illustration, an ideal solution to the 'predicate method' problem given above would be (in natural-language form) "The method must return true if all clauses are true, with the only clause being the method argument length must be no greater than 20 characters long". From this ideal solution, we know to look for a con-

dition that matches the clause and has the right effect on the value returned from the method.

Essentially, it must accurately represent the problem text, with no leeway on either side; an ideal solution that was too loose would let through incorrect solutions, whereas an ideal solution that was too tight would reject correct solutions. The constraints will use this ideal solution to check against the student solution to confirm that a requirement is being fulfilled.

4.5.2 Semantic Constraints

Semantic constraints are developed by analysing individual solutions to problems, and generalising. As the semantic requirements of a problem are represented by the ideal solution, each semantic constraint will first look at the ideal solution to see if it is relevant for this problem. An example of a semantic constraint (written in natural-language) is shown in Figure 4.5. This constraint would be relevant for a problem such as the iteration problem in Section 4.3; for this problem, we require the student to iterate between the two arguments (`startNum` and `endNum`), and square each number along that path of iteration, saving each squared value into a running total. For this task, we need a loop, which is what the constraint is checking for.

```
IF    a problem requirement is to apply a function
      to a range of numbers
THEN  it must be the case that the solution contains
      a loop
```

Figure 4.5: Semantic constraint example

CHAPTER 5

Implementation

The J-LATTE system is implemented using the client-server metaphor. The user accesses the system through his or her web-browser; a browser with the J-LATTE website accessed and the J-LATTE software loaded into memory is considered the client, and this connects through web transactions such as problem-loading and solution-submission to a server, which does the bulk of the system processing like keeping student records, diagnosing solutions, and deciding what feedback to display. After discussing the system architecture, this chapter details the implementation through three views; the client software, the server software, and the student's view of the system (the interface).

5.1 Architecture

The system architecture of J-LATTE adheres closely to the architecture of other constraint-based tutors (see Figure 5.1). All information and interaction is presented to the student through a web interface, which can be viewed in any mainstream web-browser. The session manager handles any requests from the web-server. It works as

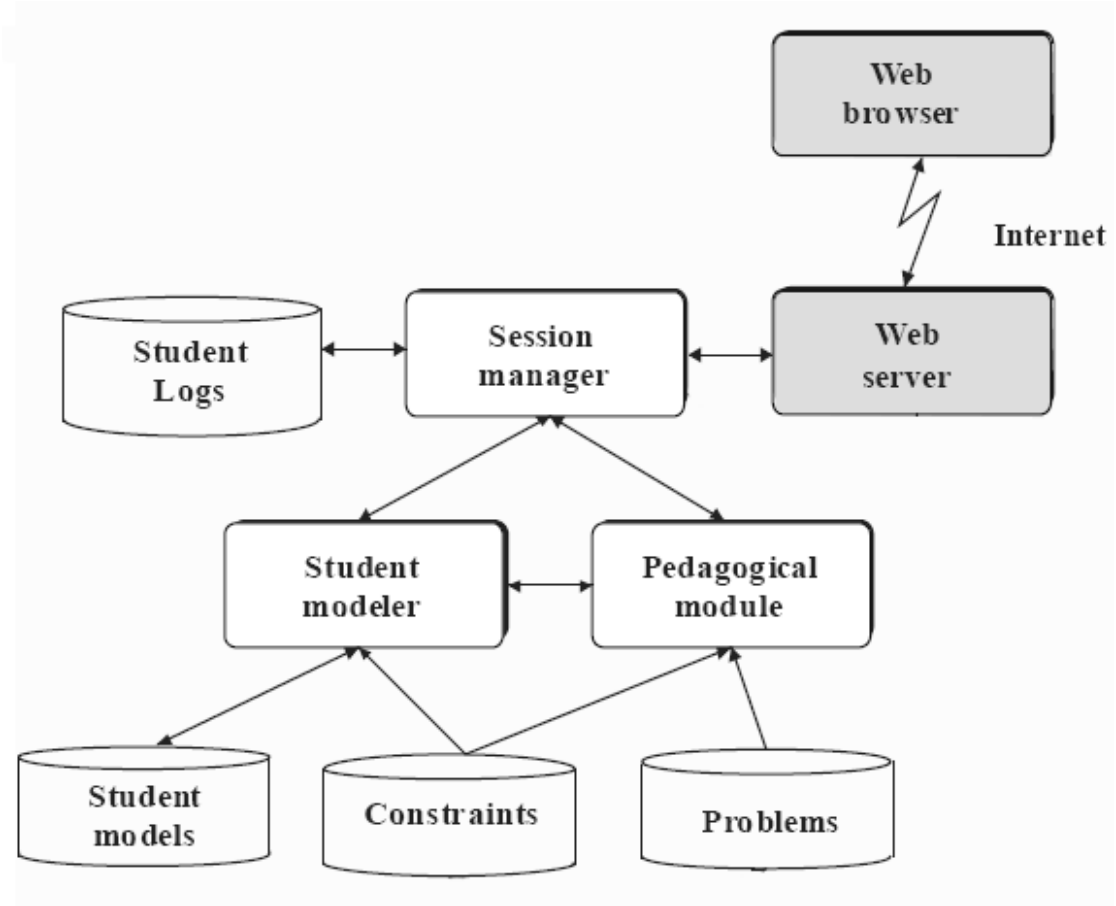

Figure 5.1: System Architecture of J-LATTE

a hub, and interacts with most other parts of the system at some point. Pedagogical decisions and operations take place in the pedagogical module (PM). This receives the interactions (via the web-server and the session-manager) from the student, such as problem selection and solution submission.

5.2 Problem Solving Interface

The user's interactions with the system occur in the problem-solving interface, a dynamically-generated web-page illustrated in Figure 5.2. Via this interface, the student

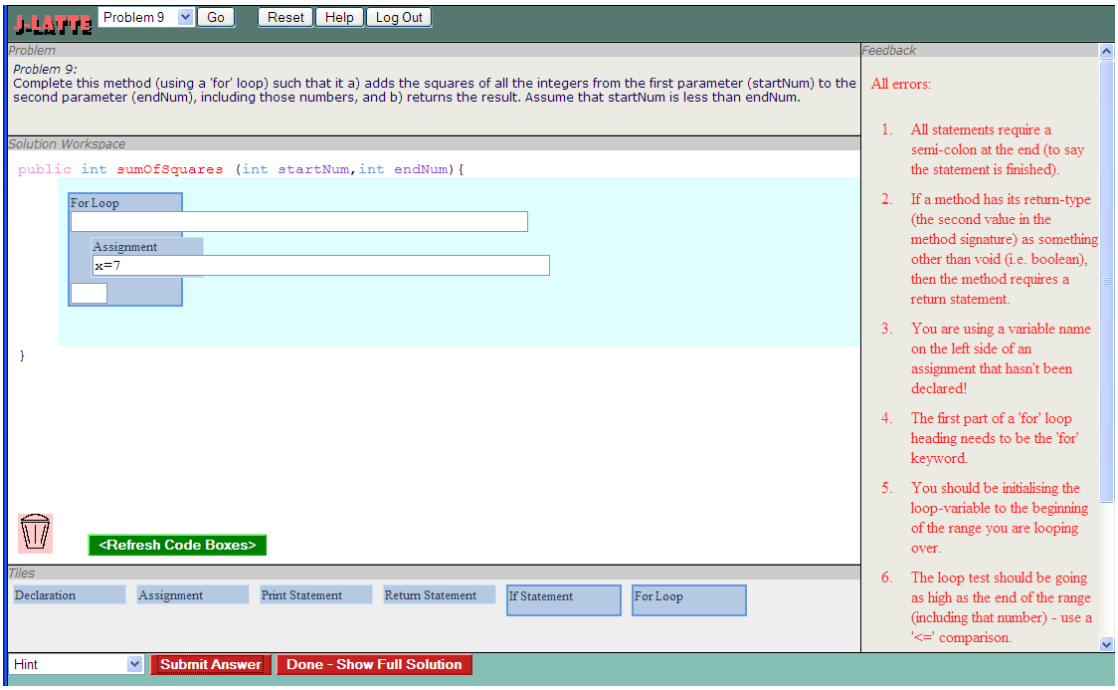

Figure 5.2: J-LATTE Interface Layout (Coding Mode)

is able to perform all necessary interactions, aside from logging in, which is handled on a separate page. These include operations such as problem selection, problem solving, and submitting a solution. The screen is split into several panes, with four main panes being related to tutoring activities. The top pane presents the problem text to the student, while the large middle pane is the solution workspace and allows the student to form a solution to the given problem. The bottom pane contains components (tiles) to be used in solution formation, and the rightmost pane presents feedback on a student's solution.

The interface (and the general layout of the task) follows the 'concept abstraction' design, such that the system helps the student to handle the complexity of a program, by first presenting the student with the concept abstractions in the form of *tiles*. The two modes of interaction as mentioned in Chapter 4 are represented here by either forming a

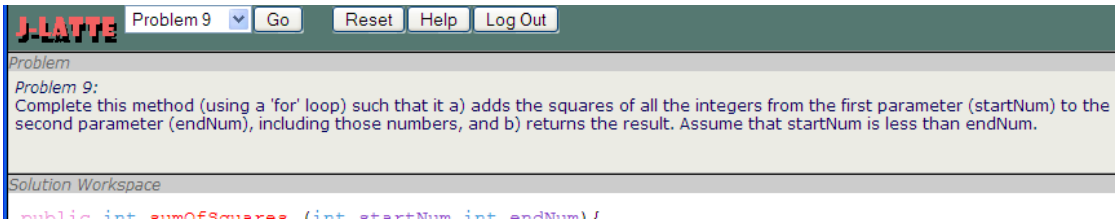

Figure 5.3: Interface - Problem Description

solution outline using these tiles (Concept Mode), or by entering code-text into the tiles (Coding Mode).

This section details all the relevant interface and interaction concepts. The general layout of the interface was chosen as it is similar to other successful constraint-based tutors developed by the Intelligent Computer Tutoring Group (ICTG), at the University of Canterbury, New Zealand.

5.2.1 Loading a Problem

The student selects the problem through the topmost pane. A problem can be selected from the drop-down box, and following this the 'Go' button can be pressed to load the problem into the client and subsequently the interface.

After selecting a problem, the student will find the problem text within the Problem Description Pane. This pane has the title "Problem" and is illustrated in Figure 5.3. It contains the description of the currently selected problem, which is displayed as text, with any stylistic variations (such as emphasis and lists) handled by XHTML rendering (XHTML tags may be inserted into the text that is stored on the server).

The problem description lists what is required of the solution for this particular problem, and should be read carefully by the student.

5.2.2 Solution Formation

Solution formation involves using the tools of the interface to create a solution to the given problem. A complete solution takes the form of a set of *tiles*, representing the Java concepts noted in the Chapter 4. Each tile represents a different concept, and the concept name is displayed as text within the tile. There are two types of tiles, corresponding to the conceptual types: statement tiles and block tiles. Statement tiles map to individual Java statements, such as 'Return Statement' and 'Assignment', whereas block tiles map to recursive concepts that can contain other statements and blocks, such as a 'For Loop' or an 'If Statement'. These block tiles can have other tiles nested inside them. Block tiles are distinguishable from statement tiles by a larger size and a darker border.

These tiles can also have text fields embedded inside them; these text fields are where a student can enter actual Java code. The initial tile state is not to have an embedded field, but when the 'Refresh Code Boxes' button (see Figure 5.2) is clicked by the user, any tiles that are part of the current solution will have a text field inserted inside (if there is not one already). These text fields can be edited immediately by clicking inside them and typing; once a student clicks inside a text box, they have entered coding mode. A statement tile contains one text field, corresponding to a single line of code, whereas a block tile contains two text fields: one for the block header (everything up to and including the first brace, i.e. "`if(x==7){`"), and one for the block footer (for all current block tiles in J-LATTE, this is just an ending brace). Any nested tiles go between the text fields in a block tile. The reason for making the student explicitly decide to enter coding mode is so that they are encouraged to think about the higher-level solution first.

Certain block statements, such as `for` and `if`, have non-block equivalents in Java; in these cases, there would be no start or end brace, and the bodies of these elements can only contain a single statement. In our system, we have not allowed the student to use

these equivalents, due to possible confusion due to ambiguity, especially for novices. It is considered by some to be good practice to not use the non-block version in real code. Also, the decision to make the student enter the end brace, even though an end brace will always be the element to be entered, is consistent with our goal to reduce the possibility of transfer problems by having the student enter a complete program.

Two panes are relevant in solution formation: the Solution Workspace, and the Tile Pane. The solution itself is formed inside the Solution Workspace; a solution outline is created by dragging tiles from the Tile Pane.

Solution Workspace

The Solution Workspace area is where the solutions to the various programming tasks are formed. This involves dragging tiles from the Tile Pane to 'input areas' (light-green blocks) within the workspace. Tiles can be moved within or between 'input areas' (if there are multiple input areas in a solution), or nested inside Block tiles, as the student sees fit for a problem. Solution tiles can also be deleted by dragging and dropping a tile into the trash icon in the bottom-left of the solution workspace. The solution workspace can also present code fragments known as "context", which frames a problem for the student (such as a method definition).

Figure 5.4 shows the initial 'clean' state of the workspace, whereas Figure 5.5 shows the workspace with a solution formed inside. The system supports moving between text fields using the arrow keys. This has been implemented to mimic a typical programming editor's behaviour of using the arrows keys to move between lines, in addition to the traditional web-browser text field navigation behaviour, which is to use either the mouse to select a text field, or use TAB and SHIFT-TAB to cycle through the text fields.

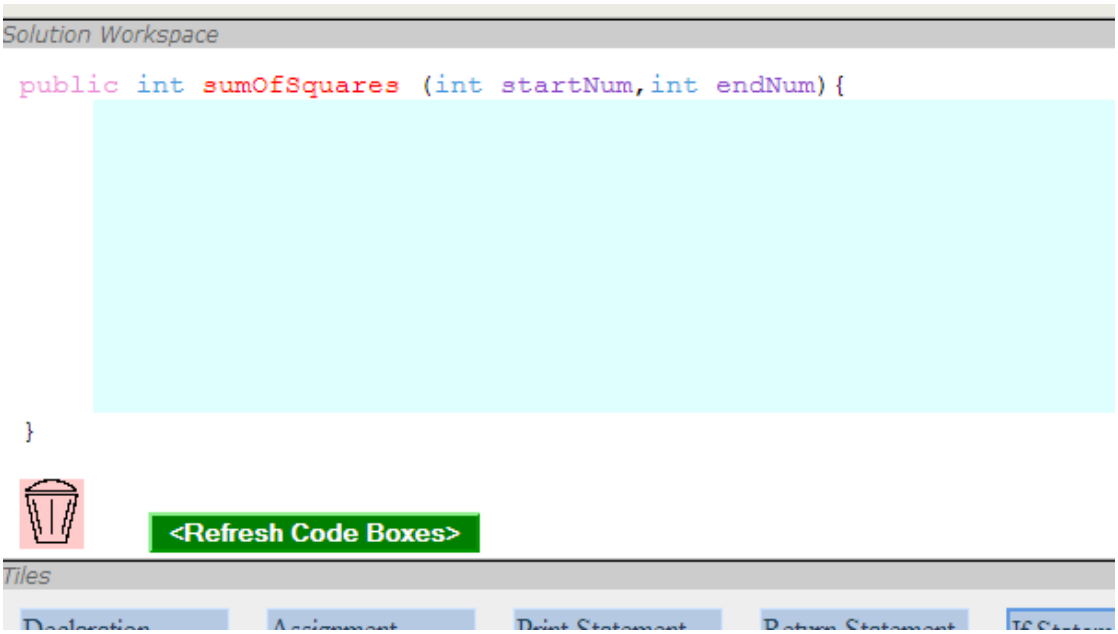

Figure 5.4: Solution Workspace (initial state)

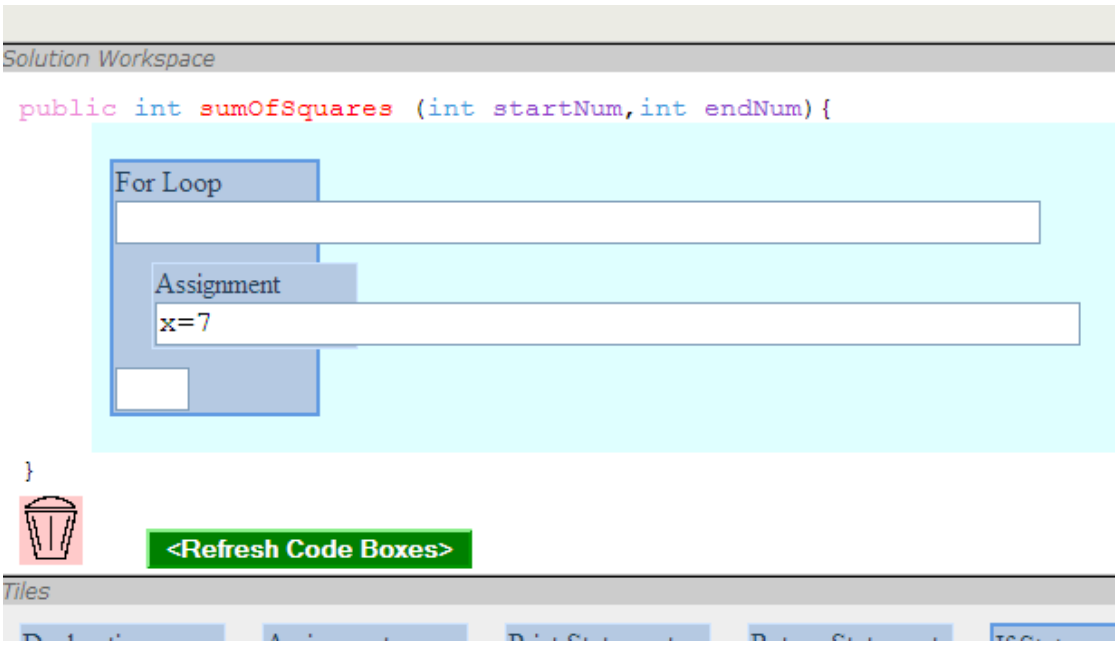

Figure 5.5: Solution Workspace (with formed solution)

Figure 5.6: Interface - Tile Pane

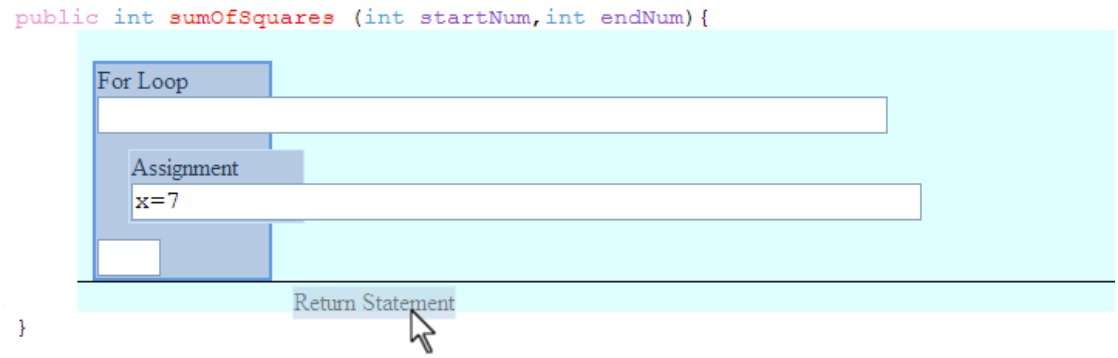

Figure 5.7: Interface - Dragging Action.

Tile Pane

The Tile Pane, with the title “Tiles” (illustrated in Figure 5.6), is where tiles that represent Java concepts are initially made available. These tiles are used by the student to form the solution using the aforementioned tile dragging-and-dropping.

Tiles taken from the Tile Pane are treated as copies; the original tile remains, and many copies can be created through the act of dragging-and-dropping. This allows a solution to have potentially infinite instances of any programming concept.

As a visual aid, when a drag-and-drop action is taking place, a transparent version of the tile will be placed at the mouse cursor to indicate this. As the tile is being dragged near an input area or block tile, a horizontal black line will appear to give an indication of where the tile will be placed if it is ‘dropped’. The tile is dropped by releasing the mouse button. Both the line and the transparent version will disappear once the tile is dropped. An example of this action is shown in Figure 5.7.

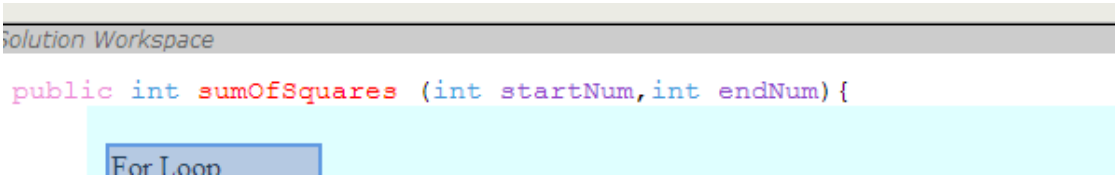

Figure 5.8: Interface - Context

Context

Figure 5.8 shows how a problem's *context* (code fragment that frames the problem) is displayed. For the sake of real-life coding experience authenticity, it is displayed using a monospaced 'typewriter'-style font, similar to that found in some programming editors; this font is also used for the code-text that is entered into the tiles by the student. Also, syntax colouring is implemented using colours typical of programming editors. As an example of its usage, in J-LATTE, argument types receive a light-blue colour, whereas method names receive a red colour. This syntax colouring does not extend to the code inside a tile's text field, due to technical difficulties; colouring individual words inside an HTML text field that is being edited is difficult to implement.

5.2.3 Solution Submission

When a student wishes to evaluate a solution that they have created, the Submit Answer button can be clicked (see Figure 5.2), and the solution will then be evaluated. After J-LATTE has evaluated the solution, feedback relevant to the submitted solution is displayed. The student can use this information to discern what is incorrect about his or her solution, and make suitable changes. Two panes are relevant for solution submission: the Submission Pane and the Feedback Pane.

Submission Pane

The Submission Pane (at the bottom of Figure 5.2) contains buttons related to submitting a solution. Inside there are two buttons and a drop-down box. The drop-down box allows the student to choose a feedback level before submitting. The Submit Answer button will cause the solution to be submitted and evaluated and feedback will be given. The "Done - Full Solution" button will pop-up a window showing a correct solution (static image) to the current problem.

Feedback

The feedback for the most recently submitted solution is displayed in the Feedback Pane (the rightmost pane), illustrated in Figure 5.9. Before submitting, a student can select the feedback level. There are three possible levels: Simple Feedback, Hint, and All Errors. Simple Feedback will only show whether the solution is correct or not, Hint will only show the first error, and All Errors will display a list of all the errors in the student's submission. If a student's submission is correct, then the student will be congratulated via the feedback (with green text instead of red), and they will be asked to choose another problem.

As well as students being able to select their own feedback level, the feedback level changes automatically, after each submission, according to a rule. The level will be reset whenever a new problem is started. Initially, the level will be set at Simple Feedback. After one submission for a problem, it will move automatically to Hint. For every following submission for that problem, the feedback returned will be for whatever level the student chose, but once the feedback is returned and displayed, the level will be reset to Hint.

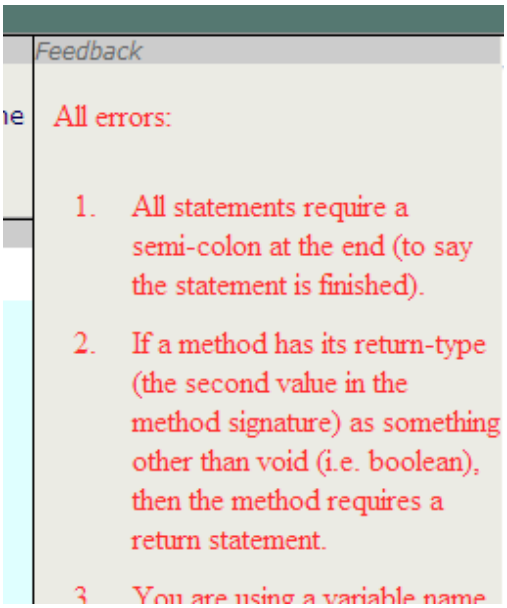

Figure 5.9: Interface - Feedback Pane

5.2.4 User Interaction Process Summary

The interaction process is as follows (ignoring login and logout). The student is presented with the problem-solving interface. The first problem will be loaded into the interface, which entails displaying the problem text and the context, and clearing the solution workspace and feedback panes. A student reads the problem, and considers the problem requirements, then starts to form an outline of the solution. To form this outline, a student drags tiles from the tile pane into the solution workspace area, arranging them as he/she believes the abstract solution to be. Once the student has reached a stage where they want to write code, they then click the 'Refresh Code Boxes' button to insert text fields within all the tiles. The student then enters Java code into these fields, further attempting to complete a solution that matches the problem requirements. If the student wishes to amend the outline, either through the extension or removal of components, this is also possible, even after code boxes have been inserted. When a student has either felt that their solution is complete, or they are unsure about how to

continue, they can click the Submit Answer button. The system is flexible: the student can perform the submission even with only a purely abstract solution. The solution will be evaluated by the server, and feedback will be returned and displayed. The student can then use this feedback to decide on how to further complete the solution. This solution forming/submission loop continues until the solution is correct. The student may then select another problem to work on.

5.3 Client-Server Overview

The client has been implemented as a set of dynamic web pages. The interface is written using AJAX technologies, which is the combination of Dynamic XHTML, JavaScript, and the asynchronous request object. AJAX allows dynamic pages to be generated inside a modern web browser, and also allows interaction with the server that does not require a page refresh. As a result of these properties, AJAX has given rise to 'Web Applications' - websites that mimic the look, feel, and behaviour of desktop applications. J-LATTE uses AJAX to implement the drag-and-drop functionality, and to transmit problems from, and student-solutions to, the server. All other programming is done using basic XHTML and JavaScript.

The web server software used for the server implementation was AllegroServe[1], a LISP-based web server running on the Allegro Common Lisp[2] (ACL) platform. All of the server-side code was written in ACL; this influenced how we transferred data between the server and the client (we receive solution submissions from the client as valid LISP S-Expressions encoded within POST requests).

[1] http://allegroserve.sourceforge.net/
[2] http://www.franz.com/

5.4 Client

This section details the parts of the J-LATTE client, which is the software that the user runs inside their web browser. We will look at how solutions are represented internally, how the drag-and-drop tile manipulation was programmed, and the client-side solution-submission behaviour.

5.4.1 Internal Student Solution Representation

The internal representation of the student solution is different for the client and the server, but conceptually they are identical - each one maps very closely to the visual representation of the solution, i.e. a tree-like structure rendered in the implementation language of choice. In both cases, we have Statement and Block elements - a Statement's only property is its code-text, whereas a Block can have two lines of code-text and a set of ordered child elements (either Blocks or Statements). The only differences between the visual structure and the internal structures are top-level annotations such as problem number and usercode (and then, only on the server).

On the client-side, the state of the solution is represented by the XHTML itself; that is to say, the XHTML code, which is rendered to create the visual solution specified by arrangement of the tiles and code, *is* the solution. There is no custom solution structure manually being built-up in memory by the client; any XHTML code is naturally represented internally in JavaScript as an object, allowing us to easily parse the code through the DOM tree when necessary. This feature of JavaScript gives the system a robust object-based solution representation without any extra processing from J-LATTE. By keeping the relationship between the interaction and the solution itself atomic, we reduce the risk of inconsistencies arising by maintaining two closely-related structures.

A downside of this arrangement is that we are limited to annotating our structure by what is possible using XHTML tags, therefore the client's solution representation cannot potentially be as rich as a custom object representation; this richness though was not required for the chosen design of the system.

The XHTML tags used in the student solution are the generic <DIV> and <SPAN> elements. These are considered non-stylistic, unlike style elements such as <B> (for emboldening text) and <font> (for selecting font properties like size and typeface), and are often used in modern website design to represent semantic structure separately from style (although in some web documents that utilise <DIV> and <SPAN>, this methodology is not always followed). Cascading Style Sheets (CSSs) are then written to style the elements appropriately. This results in a document whose look can be changed independently from its physical structure. In this system, a set of test solutions in XHTML were created, and the look was able to be tweaked easily without modifying the solutions themselves, and only the document's CSS.

<DIV>s are used for line-terminated elements, such as statements and blocks. <SPAN>s are used to encode the sub-elements of these <DIV>s in the context, such as the method-scope and the parameter types.

Each <SPAN> is given a semantic 'class name', whereas each <DIV> is given a printed name (that the student will see) appropriate for the Java concept that it is representing. As a <DIV> does not need to be styled differently from any other <DIV> (the only distinction is between Statements and Blocks), there is no need to have a Java-based class name.

An example of a student solution, as it looks in XHTML code, is shown in Listing 5.1; this listing represents the solution shown in Figure 5.10. This code in the listing has been simplified to leave only the semantic information (event code to handle other interface behaviour has been removed).

```
<div id="work_pane_work_area">
        <span class="program"><span class="jmethod">
        <span class="method-scope">public</span> <span class="method-return-type"
            >boolean</span>
        <span class="method-name">isValidName</span>
        <span class="method-params">(<span class="param"><span class="param-type"
            >String</span> <span class="param-name">name</span>
        </span>)</span>{<span class="method-body">
          <div class="jinput">
            <div class="j_tile">Declaration<input class="input" value='boolean
                testsPass=true;'></div>
            <div class="j_containment_tile j_tile">If Statement
                <input class="input input-beginning" value='if(name.length()<5){'>
              <div class="j_tile">Assignment<input class="input" value='testsPass=
                  false;'></div>
              <input class="input input-end" value='}'></div>
          <div class="j_tile">Return Statement<input class="input" value='return
              testsPass;'></div></div>
          </span>}</span>
        </span></div>
</div>
```

Listing 5.1: The client representation of the student solution (XHTML) for Figure 5.10

5.4.2 Tile Manipulation Implementation

Manipulation of the tiles is handled by an external AJAX-based library, Dojo[3], which bestows drag-and-drop capabilities upon selected HTML elements. The library was modified slightly to include a J-LATTE-specific callback whenever an item was dropped, so that any elements from the tile-pane that were moved could be identified and replaced with new clones.

There are two concepts to understand when programming a drag-and-drop interface with Dojo; draggable items and drop-zones. A draggable item is an item that can be picked up by the mouse cursor and moved around the page. They can be moved anywhere, but can only be placed again in registered drop-zones, which are elements that

[3]http://dojotoolkit.org/

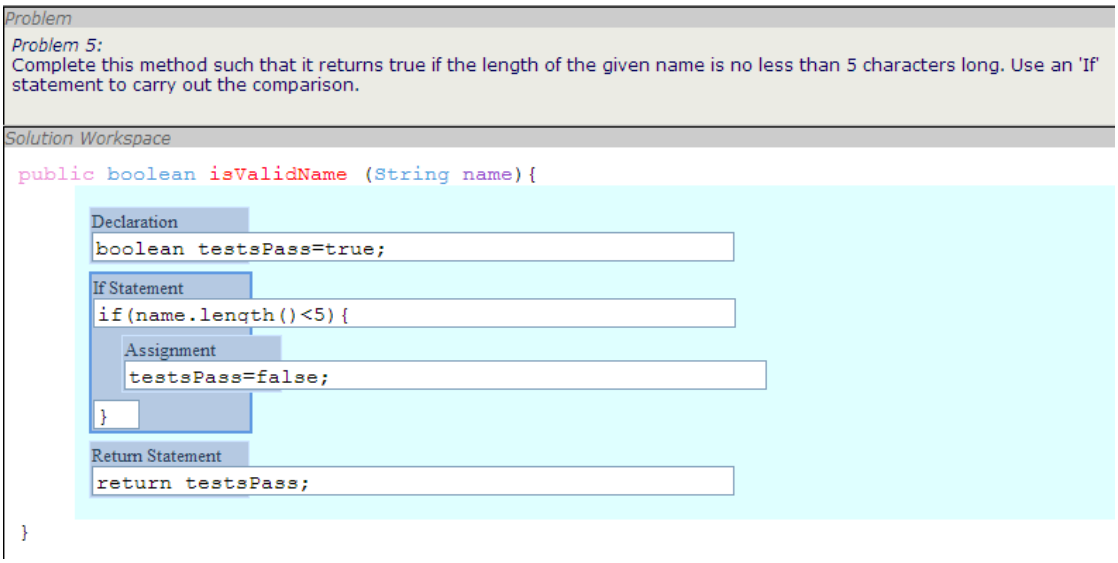

Figure 5.10: Complete solution for "Predicate" problem in Listing 5.1

can accept draggable items to be placed within. To specify a tile element to be of a particular drag-and-drop persuasion, it needs to be registered with Dojo using a simple JavaScript function. More importantly, blocks are registered as both draggable and as drop-zones, to allow them to have items inside them, as well as to be able to be placed inside other Blocks. Input areas are just registered as drop-zones, whereas Statements are registered only as draggable. At the time of the implementation of J-LATTE, Dojo was the only drag and drop library that allowed an element to be simultaneously draggable *and* behave as a drop-zone, allowing infinite nesting.

5.4.3 Solution Submission

When a solution is submitted to the server, as the server is written in LISP, the XHTML solution is parsed into a server-friendly LISP S-Expression representation. As the solution is XHTML, which is just a representation of HTML in XML, we are able to use JavaScript's inbuilt XML parser to traverse the DOM tree node by node, building a nested list as we go along. We also annotate each Statement or Block list within the

generated S-Expression with a unique ‘component-id’ property, to distinguish otherwise identical-looking components from each other. Once generated, the solution is sent as a POST request.

The other option considered as the transfer data format was XML, a popular choice as a generic data format for interfacing between two incompatible systems (both in web development and general software). Conceptually the two formats are similar; in fact, the arrival of XML heralded calls that it was the “poor man’s S-Expression”, although in reality, neither is clearly better than the other, as each has its advantages, and these differences are more minor convenience and clarity issues. Inside the client, XML is used, whereas inside the server, S-Expressions are used, so it makes sense to use one of them for transfer of solutions. In this situation (transferring submissions to the server), S-Expressions were chosen as the transference format over the more common XML format as it is simpler to parse XML into the S-Expression format in a web browser (due to JavaScript’s natural XML parsing capabilities), than it is to parse XML to S-Expressions within ACL.

5.5 Server

This section discusses the software implemented on the server side. We will look at the problem representation, with a closer look at the implementation of the context and ideal solution associated with a problem. Following this will be a description of the student solution representation, the constraint language, and the solution evaluation behaviour, as well as detailing the functionality of the student modeller.

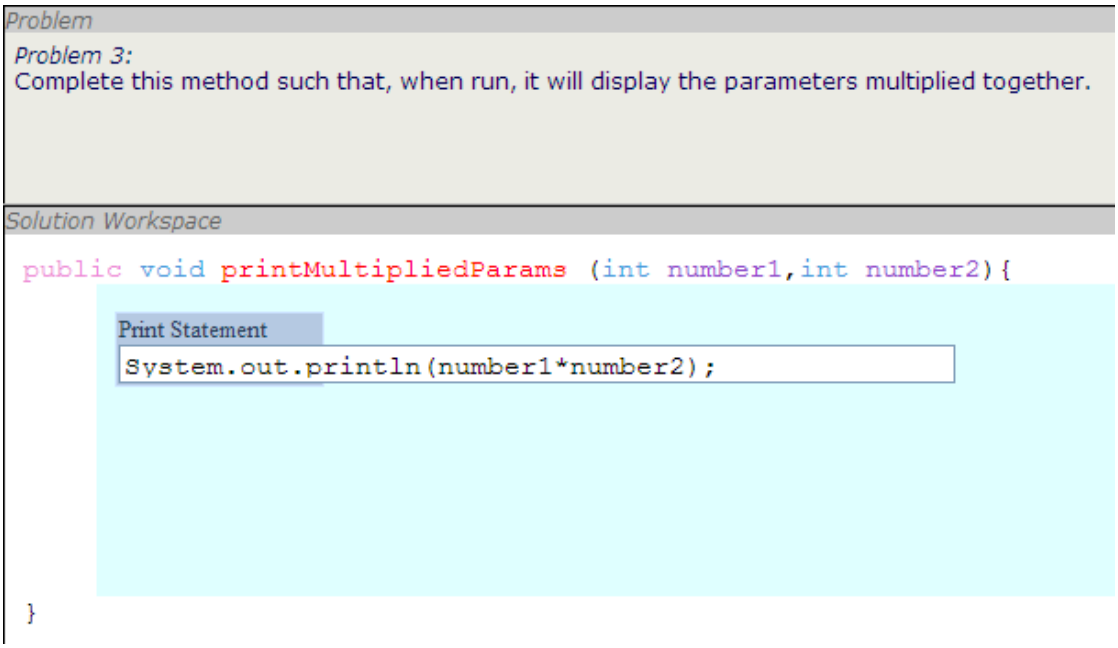

Figure 5.11: Complete solution for “Printing” problem

5.5.1 Problem Representation

The problem representation within the server is that of a similar format to the solutions. An example is shown in Listing 5.2, which is the representation for the “Predicate” problem shown in Figure 5.10. Each problem is defined by three separate sections: the problem text, the context, and the ideal solution. The problem text is encoded simply as text, although XHTML tags can be embedded to add emphasis to content. In total, 13 problems were written for the system (but at the most, 11 were in the system at one time).

As an example of problem representations for the rest of the problem types, Listing 5.3 shows the representation for the “Printing” problem in Figure 5.11, and Listing 5.4 shows the representation for the “Iteration” problem in Figure 5.12.

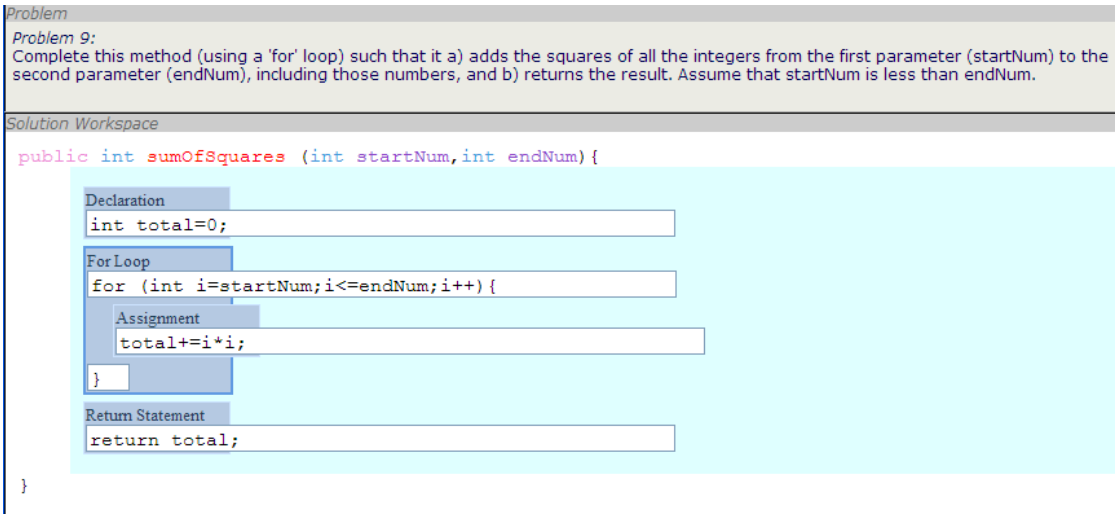

Figure 5.12: Complete solution for “Iteration” problem

```
(5
 :text "Complete this method such that it returns true if the length of the
     given name is no less than 5 characters long. Use an 'If' statement to
      carry out the comparison."
 :context (("method"
            :name "isValidName"
            :params ((:type "String" :name "name"))
            :return-type "boolean"
            :scope "public"
            :body (("input"))))
 :ideal-solution ((test method-argument-length :type (>= 5))))
```

Listing 5.2: The internal problem representation for the “Predicate” problem in Figure 5.10

Context

The context section of the problem contains information about the part of the Java code that will be given to the student when the problem is chosen (represented as LISP plists). The representation of the context shown in Figure 5.8 is shown in Listing 5.12. Conceptually each major Java element has a set of properties: for example, a method would have a *name* and a *return type* (amongst other things), a declaration would have a *type* and an *expression* etc. Each element in the context contains a tag as its head, naming

```
(3
 :text "Complete this method such that, when run, it will display the
    parameters multiplied together."
 :context (("method"
             :name "printMultipliedParams"
             :params ((:type "int" :name "number1")
                      (:type "int" :name "number2"))
             :return-type "void"
             :scope "public"
             :body (("input"))))
:ideal-solution ((output (multiplication args))))
```

Listing 5.3: The internal problem representation for the "Printing" problem in Figure 5.11

```
(9
 :text "Complete this method (using a 'for' loop) such that it a) adds the
    squares of all the integers from the first parameter (startNum) to the
     second parameter (endNum), including those numbers, and b) returns
    the result. Assume that startNum is less than endNum."
 :context (("method"
             :name "sumOfSquares"
             :params ((:type "int" :name "startNum")
                      (:type "int" :name "endNum"))
             :return-type "int"
             :scope "public"
             :body (("input"))))
 :ideal-solution ((sum-of-function-over-a-range
                   :range (:from (method-arg :name "startNum")
                           :to (method-arg :name "endNum"))
                   :function square)))
```

Listing 5.4: The internal problem representation for the "Iteration" problem in Figure 5.12

the concept, followed by a LISP property key/value pair (i.e. `:name` "isValidName") for each property. This is done to simplify the solution evaluation process - by naming each part of the code now, we don't need to parse each element later to discover its syntactic meaning. The interface also uses this encoding to highlight the various components of

the context. The '`("input")`' element informs the client where to allow the student's input (i.e. where they can drag tiles and add code).

Ideal Solution Representation

The ideal solution is also written in a custom language, encoded as a LISP S-Expression. The ideal solution contains the abstract requirements of the solution, stated succinctly and with arguments to allow customisation to the problem at hand. The ideal solution towards the end of Listing 5.2 details the requirements for the "Predicate" problem given in Figure 5.10. The semantic constraints read this language to decipher what is required, and search for the satisfaction of these requirements inside the solution. This particular segment tells us that we have a test the solution should be carrying out (note that it is possible to have more than one test for a problem), therefore it is a predicate method, and the test is that the method-argument length should be greater than or equal to 5. The constraints will look for an equivalent test to this, taking into account transference of values through variables. As an example of this flexibility, for the test `arg>=5`, instead of checking directly against the argument variable, if the argument's value has been assigned to any other variable (for example `myX`), then we can instead use that variable in place of the method argument variable when doing the comparison. Also, an equivalent test could be `!(myX<5)`, which checks the same condition but in a different way. Due to the complexity of programming and the varied ways a clause can be written, not all possibilities were able to be considered for the final system, but the constraints were general enough to account for the most common situations. Further generalisation of the semantic constraints would provide a better consistency with the Java standard.

Another ideal solution example is for the "Iteration" problem described in Listing 5.4. The ideal solution here is telling the system that we require the student to loop between the values of the two arguments, and square each number. The parameters to

this ideal solution can be changed, such that the `:from` or `:to` properties can instead reference literal values (like '1' or '22'), if that's what the problem requires.

The ideal solution language itself is not as general as originally intended; each construct is more specific than would be preferred (for example, `sum-of-function-over-a-range` is too low-level to have a wide use over many problems). Designing such a language is difficult, but with more time the language could be revised to be more general. The grammar of the ideal solution language is shown in Figure 5.13.

5.5.2 Student Solution Representation

The solutions that are submitted from the client are converted before POST-ing from XML to a semantically identical version as S-Expressions, except for annotations such as the current problem number and feedback level. An example of the representation for the solution in Figure 5.10 is shown in Listing 5.5. Notice how the part that refers to the context is in the same form as it was originally sent from the server to the client during problem loading (apart from the removal of input areas and the addition of component-id properties to make components unique). As a further example, Listing 5.6 and Listing 5.7 show the solution representation for "Printing" and "Iteration" problems, respectively.

Tiles in a submitted solution are represented in a similar way to the context (the overall structure is the same), but the only properties a tile contains are the component-ids, a flag registering it as a tile, and 'input text' properties. These 'input text' properties contain the code that was entered for a tile. A statement tile contains an `:input` property, whereas block tiles contain `:input-beginning` and `:input-end` properties to repre-

```
Ideal Solution:
(output OutputBody)|(test TestBody)+|
(sum-of-function-over-a-range SumBody)|
(count-value CountBody)
```

```
OutputBody:
[argument|(Operation args)]
```

```
Operation:
[multiplication|addition]
```

```
TestBody:
([MethodArgumentTest|MethodArgumentLengthTest])
```

```
MethodArgumentTest:
method-argument :type (ArgTestType)
```

```
ArgTestType:
[must-not-contain character|not-empty]
```

```
MethodArgumentLengthTest:
method-argument :type (LengthTestType)
```

```
LengthTestType:
[Comparator|number]
```

```
Comparator:
[=|<|>|<=|>=]
```

```
SumBody:
:range (:from RangePart :to RangePart)
:function Function
```

```
RangePart:
[NumberRange|MethodArgRange]
```

```
NumberRange:
(number :number number)
```

```
MethodArgRange:
(method-arg :name argument-name)
```

```
Function:
[identity|square]
```

```
CountBody:
:of MethodArgRange
:to-count [(literal :char character)|MethodArgRange]
```

Figure 5.13: Ideal Solution Grammar

sent code occurring at the beginning and end of the block tile. Block tiles also receive a 'body' property, in which child elements are listed in order.

```
(:problem-id 5 :fb-level simplefb :code
    (("method"
      :component-id 1 :scope "public" :return-type "boolean" :name "isValidName"
      :params ((:type "String" :name "name"))
      :body (("declaration" :component-id 2 :tile "t"
               :input "boolean testsPass=true;")
             ("if-statement" :component-id 3 :tile "t"
              :input-beginning "if(name.length()<5){"
              :body (("assignment" :component-id 4 :tile "t"
                      :input "testsPass=false;"))
              :input-end "}" )
             ("return-statement" :component-id 5 :tile "t"
              :input "return testsPass;"))))
```

Listing 5.5: The server representation of the student solution for the "Predicate" problem in Figure 5.10

```
(:problem-id 3 :fb-level simplefb :code
    (("method"
      :component-id 1
      :scope "public"
      :return-type "void"
      :name "printMultipliedParams"
      :params ((:type "int" :name "number1")
               (:type "int" :name "number2"))
      :body (("print-statement"
               :component-id 2
               :tile "t"
               :input "System.out.println(number1*number2);")))))
```

Listing 5.6: The server representation of the student solution for the "Printing" problem in Figure 5.11

```
(:problem-id 9 :fb-level simplefb :code
    (("method"
      :component-id 1
      :scope "public"
      :return-type "int"
      :name "sumOfSquares"
      :params ((:type "int" :name "startNum")
               (:type "int" :name "endNum"))
      :body (("declaration" :component-id 2 :tile "t"
              :input "int total=0;")
             ("for-loop" :component-id 3 :tile "t"
              :input-beginning "for (int i=startNum;i<=endNum;i++){"
              :body (("assignment" :component-id 4 :tile "t"
                      :input "total+=i*i;"))
              :input-end "}")
             ("return-statement" :component-id 5 :tile "t"
              :input "return total;")))))
```

Listing 5.7: The server representation of the student solution for the "Iteration" problem in Figure 5.12

5.5.3 Constraint Representation

Constraints are represented and stored as LISP lists on the server; examples of this representation are shown in Listings 5.8 and 5.9, which list the representations for a syntax and a semantic constraint. The first element of each constraint is the id of the constraint. To make development easier, each constraint is given an id that describes its general concept ('decl' for constraints that handle declarations, 'return' for ones that handle return statements etc), followed by a number to make each constraint id unique. These numbers are generally sequential, although some constraints have hyphenated numbers (e.g. "decl1-2") as a way to group sub-concepts (this has no bearing on the effect of the system, it is purely to aid constraint organisation during development). The `:feedback` property contains the string that will be displayed to the student if this constraint is violated. The `:type` property details whether the constraint is syntactic,

semantic, or style, but is not used by the system in any way; it is only a categorisation that may be useful for the developer. In total, 89 constraints were implemented in the system; 44 syntax, 43 semantic, and 2 style.

```
'(decl2
  :feedback "The first part in a declaration must be a valid Java type."
  :relevance ;; for all declaration tiles which aren't empty and have no equals sign
  (and (not (solution-empty-p ss-code))
       (bind-all ?decl-tile (find-all-tile-elements-of-type "declaration"
           ss-code) bindings)
       (not (tile-input-empty-p ?decl-tile))
       (match '(?first ?second ?*rest-decl ";") (java-tokenise-string (
           get-component-input ?decl-tile)) bindings)
       (not (equalp "=" ?first))
       (not (equalp "=" ?second)))
  :satisfaction ;; must be a valid type
  (valid-java-type-p ?first)
  :type syntax)
```

Listing 5.8: An example syntax constraint

```
'(sum-of-function-over-a-range5
  :feedback "You should be initialising the loop-variable to the beginning
      of the range you are looping over."
  :relevance ;; is sum-of-a-function
  (and (not (solution-empty-p ss-code))
       (equalp (first (first is-code)) 'sum-of-function-over-a-range)
       (bind ?iteration-tiles (find-iteration-tiles ss-code) bindings)
       (bind ?loop-tile (first '?iteration-tiles) bindings)
       (not-nil ?loop-tile))
 :satisfaction
 (loop-var-initialised-to-p (first ss-code)
                            (get-range-arg-name
                            (getf (getf (rest (first is-code)) :range)
                            :from) ss-code)
                            ?loop-tile
                            (get-loop-var-name ?loop-tile))
 :type semantic)
```

Listing 5.9: An example semantic constraint

The `:relevance` and `:satisfaction` properties represent the relevance and satisfaction conditions of the constraint. Each condition is written in a custom constraint language, which is based upon LISP code, and was originally implemented in SQL-Tutor [Mitrovic and Ohlsson, 1999]. Each is evaluated to return either true or false at submission time. Each test in the condition will either refer to a domain-specific function, a variable binding, or a pattern match. Further details about the use of the conditions can be found in the Solution Evaluation section later in this chapter.

Variables and Binding

Variables are any words that are preceded by a question mark (an example is '?decl-tile'). Values can be bound to variables either through the `bind`, `bind-all` and `match` functions. `bind` is a simple binding of a value to the variable, whereas `bind-all` takes a list as an argument. For each element in this list, the evaluator forks the evaluation and creates a separate bindings list, with the list-element being bound to the given variable only for that fork. Any future references in that fork to the variable will return the bound value.

Domain-Specific Functions

Domain-specific functions perform tasks that are too complex to be encapsulated inside a constraint with the normal constraint language. Most domain-specific functions perform a test, which is to return true or false depending on their arguments. Often these arguments will be previously bound elements, or maybe even the student solution itself. The remaining domain-specific functions return elements from the solution to be bound, such as all the declaration statements, or the code from a particular tile. Within a pre-implementation constraint (i.e. written in natural-language at the design stage), can-

didates for domain functions will be other statements than those that perform a binding action, i.e. “If we have a well formed expression...”, or “...then it must be the case that a semi-colon follows the statement”. Each domain function is given a descriptive name, to aid when reading and writing the constraints in LISP form. For the two candidates just given, the tests were recorded as `(well-formed-evaluating-expression-p ?expression)` and `(component-input-finished-by-semicolon-p ?n)` (the “-p” extension is a LISP convention that refers to the function being a predicate, and the ?n, in this context, refers to a variable with the value of a code-string). The generality varies; some domain functions are specific to a particular concept (and are therefore only used in a handful of constraints), whereas others tackle a more general Java idea, and are used in a greater number of constraints. There are not many domain functions of the former case though.

5.5.4 Solution Evaluation

Solutions are evaluated by the student modeller (discussed in Section 5.5.5), which passes them through the evaluation network. The evaluation network is designed to evaluate a given solution, and return a list of violated constraints. The concept of evaluation is simple: first evaluate the relevance conditions of all constraints over the solution, sending whichever constraints pass to the second stage, where the satisfaction conditions are then evaluated; when a satisfaction condition passes, the constraint is recorded as ‘satisfied’, but when one fails, the constraint is recorded as ‘violated’ for this submission. A solution is correct when there are no violated constraints recorded. As the two conditions are written in the LISP-based custom constraint language, the system avoids the need for a complex evaluator, as many of the operations can be called

directly from LISP (although some evaluation code is there, mainly to keep track of bindings and some special conditions).

At the end of evaluation, the feedback messages are chosen according to the selected feedback level. These are returned to the client. Also, the student model (the implementation of which is described in Section 5.5.5) is updated by the student modeller.

The constraint network itself is optimised according to the Rete algorithm [Forgy, 1982]. This was already part of the evaluation engine utilised for the system, and was not implemented by the system author. The Rete algorithm essentially reduces the amount of evaluation needed to be done, by finding multiple constraints with overlapping subconditions (e.g. three constraints whose relevance conditions all begin with the same step), and taking these subconditions and attributing them to a network node (which is also assigned the numbers of the constraints these conditions are related to). Then, during evaluation, these subconditions only have to be evaluated once, but the result applies to all constraints that have these subconditions. Nodes with more of these subcondition patterns are attached as children, and the network is built up until all constraints can be represented by the network. Evaluation involves propagating through the network via each of the input nodes.

5.5.5 Student Modeller

The student modeller creates and updates, for each student, a student model, as well as evaluates student solutions. The purpose of the student model is to record the domain knowledge of the student. It does this by recording the constraints violated and satisfied by the student's submissions over time. Groups of constraints can also be linked to greater concepts, allowing us to perform analysis such as "Student Y knows 30% of concept X". The student model is also intended to be used for future problem selection.

Currently, the student model itself contains a list of all the constraints, with each constraint linked to its complete satisfaction and violation history. We derive the student's knowledge of a particular constraint by taking a window of the most recent history (up to 5 instances), and calculating a ratio of satisfied-instances to total-number-of-instances in the window. For example, if in the last 5 instances, the student has violated the constraint twice, and satisfied the constraint thrice, then the ratio will be 3/5. We regard this ratio as how much a student "knows" a constraint, although this is a naive measure; other more accurate representations are possible, but the one chosen was adequate for this project. The student model also keeps a list of the solved problems.

CHAPTER 6

Evaluation

The common way to evaluate the effectiveness of educational systems is to run an evaluation in a classroom or lab setting with students who are currently studying the curriculum that the system covers. We have followed this approach by performing three evaluations: a pilot study, a full evaluation with participants who were 5 weeks into a tertiary programming course, and a second full evaluation, with an improved version of the J-LATTE system, with participants who were 6 weeks into a later iteration of the same programming course.

This chapter describes the evaluation studies that were performed, and their results. The first section presents the pilot study, the second section details the first full evaluation study, and the third section details the final evaluation study that was run.

6.1 Pilot Study

In April 2007 we ran a pilot study with a group of 8 postgraduate Computer Science students as participants. The purpose of this study was to gain non-developer feedback, so that any major interaction faults, system bugs, and content issues (such as incomplete

constraints and confusing problem descriptions) could be identified and fixed before performing a full evaluation.

This study was appended to a think-aloud assignment that the students were already prescribed. This assignment required students to work through the problems in the system and talk through their reasoning. As all of these students had completed undergraduate degrees in Computer Science (with Java as their initial programming language), they were all (with one exception) very familiar with the workings of Java; therefore, their learning level was not tested.

Feedback was gained through a developer being present during the study sessions. Also, the sessions were recorded using both a video camera (recording the screen and the participant), and via a direct screen recording (to get a clearer screen picture than from using the video camera). An informal discussion also took place after each session to record the participants' thoughts. The think-aloud requirement of the assignment helped significantly in gathering feedback during the session itself.

6.1.1 Results

Two usability issues were discovered. The first issue was that tiles could be difficult to drag-and-drop into other tiles (it required more precision than was necessary). This was improved by tweaking the size of the tiles. The other issue was that the small size of the solution area made it difficult to work with larger solutions; the solution area could be scrolled vertically, but this was inconvenient. This was improved by decreasing the area for the tile panes and submit panes.

Some students had issues with the constraint messages, mainly due to the language used and their verbosity. Some of this was unavoidable (the complexity of the domain means that some constraints are more complex than others, therefore requiring a more

descriptive message), but in some cases the language was modified to make it more understandable.

The most obvious issue (from which the most useful feedback was received) was the incompleteness of the domain model (mainly semantically, although there were some small syntactic issues), for the Java domain subset that was covered by our problems. Some solutions that should have been accepted by the system were rejected due to the constraints being too restrictive, and therefore were covering the concepts incorrectly (or, in some cases, the domain functions that were being called from the constraints were incorrect). The constraints that were causing these problems were identified, and were redesigned to represent their concepts more accurately (to take into account a wider range of solutions). In other cases, a solution was accepted that was incorrect; this indicated missing constraints. These concepts were identified and the constraints either written or other constraints were adjusted to cover these situations.

Compared to issues with the domain-model and problems, relatively few technical issues were found for this study.

6.2 Evaluation Study 1 (August 2007)

In August 2007 we ran a full evaluation. The participants for this experiment were volunteer Computer Science students who were currently enrolled in the second semester version of COSC121, an introductory Java programming course at the University of Canterbury. The second semester version (coded COSC121-S2), although identical in content to the first semester version, was less populated than its predecessor; as COSC121 is generally the first course to undertake when starting a Computer Science degree, and as most students start their degrees at the beginning of the academic year, the majority of Computer Science students at Canterbury will take the first semester

version. A number of students taking COSC121-S2 fall into the categories of 'repeat' students from the first semester course, or are changing from another University discipline (though these categories of students are not necessarily the majority). These factors may lead to a greater chance of recruiting novice participants for a study.

Participation was voluntary, as opposed to being run during set lab times; this was due to lack of space in the lab timetable (students were required to submit work from their regular lab sessions, so to have a session devoted to an experiment in this case would take time away from scheduled assessed work).

There are several downsides to making the experiment sessions voluntary and outside of scheduled course lab sessions. One downside is that it can be difficult to recruit a large participant base where the audience is not as 'captive', and must make the effort in their own time. Also, because that time is limited by both other courses and non-course-related activities, the length of the sessions cannot be significant (at least not without sizeable compensation).

For this study, we asked for students to be available for no more than 90 minutes. This includes the initial session administration and setup, followed by a pretest, then a maximum of 1 hour interaction time, followed finally by a posttest and a questionnaire. At the end of their session, they received compensation in the form of $5 cash, as well as $5 in vouchers to be redeemed at an on-campus cafe. The data collected in the study was anonymous; participants were given a random, non-identifying usercode to log into the system, and to link the tutor, pretest, posttest, and questionnaire data together. The participants were asked to retain their usercode, so they could ask us to remove their data at any time from the experiment.

A study could possibly be run such that students could use the system in their own time, possibly allowing more interaction time, but that kind of experimental design still relies on the initiative of the students themselves. Also, this approach loses some control

over outside influences such as textbooks, being able to guarantee that every student will complete both a pretest and a posttest, and giving instantaneous feedback if a student encounters issues with the system.

6.2.1 Recruitment of Participants

Student volunteers were recruited from COSC121 initially by announcing the study in a lecture, at the beginning of the week before the experiment was to be run. A 5-minute demo was shown to the students; basic J-LATTE interaction while partially completing a problem was presented, but no submission was made. This was intentional, as some students would be using a version of the system that would not allow solution submission or give feedback to the student. Preceding and proceeding the demo, basic information was given orally about why the study was being run, what the study would entail for participants, and what compensation would be given to volunteers. A signup sheet was also left in the lecture theatre, requesting an email address from interested students, so that they could be contacted to confirm involvement and times. The initial response was poor, with less than 5 students responding.

The class was then sent an email once again detailing the study, and asking for volunteers. This approach proved to be the most effective of all the recruitment attempts. During this week, as well as the week of the experiment itself, students were approached in their COSC121 lab sessions. Every student who noted down a usercode and a lab time was contacted later to confirm.

In the end, 15 students participated in the study. A small number is to be expected for a voluntary study for such a small class size (there were approximately 140 people enrolled in the course).

The study was run in the 5th week of lectures, where students had covered ‘Strings’, ‘If Statements’, and ‘Object Interaction’. Students had not yet practiced ‘If Statements’ in labs.

This study was approved by the Human Ethics committee, under application number 2007/66.

6.2.2 Experimental Design

The experiment was designed to test the hypothesis that our system was more effective than a classroom setting. The definition of a classroom setting to us is that a student is given a programming problem, for example on a whiteboard, after which they would try to answer it on paper, then once the student believed they were finished the teacher would give them the answer, and the students would compare that solution and their own solution attempt.

Pre and Post tests

Two tests were created to test the participants’ Java knowledge before and after using the system. The tests were designed to be of similar difficulty, but in order to eliminate any variability between the tests, some participants received test 1 as their pretest and test 2 as their posttest, whereas others received the reverse.

Each test had 5 problems (4 short answer and 1 multichoice), and can be found in the appendices. Each question covered a different section of the curriculum they had covered in class, as well as testing a different programming skill. The tests were identical in terms of skills tested, but the questions were different. The type of each question was:

- What will a given variable be set to after the code has executed? (Understanding of value transference)
- For what reasons will this method not compile? (Understanding of syntax)
- What type should the blank be filled in with? (Understanding of types)
- Into which branches will the program flow? (Understanding of conditional program flow)
- Write a line of code assigning a calculation (Basic code generation)

Group Assignment

Two types of random assignment needed to be taken into account: between the experimental and control groups, and between which test to give the participant as the pretest. These options give us four possible permutations - Experimental and Test 1 first, Experimental and Test 2 first, Control and Test 1 first, and Control and Test 2 first. The ideal situation is to have the distribution of participants across these four groups as even as possible; therefore, assignment was random, but with a constraint that the final result must have as even a distribution as possible.

The system contained 8 problems of increasing difficulty; 3 display problems, and 5 predicate problems. There were two versions of the system. Each version allowed the participant to log in, and see the problem-solving interface. A student could move through the system problem by problem, create solutions using the tiles, enter code into these tiles, and could receive a full-solution (a static image) at any time by clicking the 'Done - Show Full Solution button'. Once the full-solution has been seen for a problem, the participant is prohibited from continuing to work on that problem, and cannot come back to it in the future. A participant is warned of this during the initial session briefing,

and also when they click the *Show Full Solution* button (a confirmation pop-up dialog is shown).

The experimental version of the system additionally allowed the participant to submit a solution and receive feedback on it. Therefore, this version had made the *Submit* button available, as well as a feedback-type selector. In contrast, the only feedback the control system would give was the full solution.

Feedback Selection

For this study, the experimental group was given three types of feedback selection to choose from; 'simple feedback', 'hint', and 'all errors'. 'Simple feedback' gives a simple "correct"/"incorrect" response, 'hint' displays one of the descriptive error messages, and 'all errors' displays all of the descriptive error messages.

In addition to this, both groups could receive feedback in the form of the full-solution. As the control group was not submitting a solution and therefore not receiving fine-grained feedback, they could not choose the feedback type.

Procedure

All sessions were held in CSSE labs; by doing this, we were able to accommodate several people participating in parallel, giving participants more freedom in choosing session times without fear of clashing with other students. In total there were 7 sessions, with the minimum number of participants in a session being 1, and the maximum being 4. The sessions were held in two different lab locations, depending on what was available (the composition of these two different locations was essentially the same). Each lab had IBM-compatible computers running the Fedora Core 4 Linux operating system, with the Firefox web browser (version 1.5) installed.

A participant would sit down at a computer, log in, and run the web browser. They would then be given the information sheet and consent form and be asked to complete them. They were also assigned a non-identifiable usercode. After completing the forms, they would be given a pretest, composed of two multichoice and three short-answer questions, which they would work on for a maximum of 10 minutes (all participants completed the tests within the required time). After completing the pretest, the participant would be asked to log into the system using the given usercode. The experiment coordinator would then briefly explain how to interact with the system.

After using the system for a maximum of an hour, or completing all the problems in the system (whichever came first), the participant would be asked to log out of J-LATTE. They would then be given a posttest, with the same conditions as the first test (complete within a maximum of ten minutes), followed by a questionnaire. This marks the completion of the experiment session for this participant, and they would then receive the compensation for their time.

6.2.3 Results and Analysis

System Interaction

	Control	Experimental	Significant
No. of participants	7	8	
System interaction time (mins)	44:08 (14:20)	59:23 (16:12)	p = 0.02
No. of attempted problems	7.4 (1)	6.8 (1.3)	n.s.
No. of solved problems	1.6 (1.3)	4 (2.1)	p = 0.008

Table 6.1: Evaluation 2007: System interaction statistics (s.d. given in parentheses)

Altogether we recruited 15 participants, with 7 being assigned to the control group, and 8 being assigned to the experimental group.

A series of student–J-LATTE interaction events were recorded by the system. This included logging in and out (detailing how long the student interacted with the system for), submitting a solution, and problem selection. The results derived from these records are shown in Table 6.1.

We found that there was a significant difference between the number of solved problems for each group (t = 2.68, p = 0.008, d.f. = 13), as well as a significant difference between the system interaction time (t = 2.56, p = 0.02, d.f. = 8).

	#	**Pretest**	**Posttest**	Gain	Significant
Control	7	2.4 (1.7)	3.1 (1.1)	0.7 (1.2)	n.s.
Experimental	8	2.1 (1.8)	3.4 (1)	1.4 (0.9)	p = 0.002
Significant		n.s.	n/a	n.s.	

Table 6.2: Evaluation 2007: Pretest and posttest scores (s.d. given in parentheses)

The pre and posttest results are shown in Table 6.2. There was no significant difference between pretest scores, thus showing that the two groups are comparable.

A significant difference was found between the pretest and posttest for the Experimental group (t = 4.25, p = 0.002, d.f. = 7), but no significant difference was found for the Control group. No significant difference was found between the gains for both groups.

The solution submission logs give us insight into the mastery of constraints. As shown by the plot in Figure 6.1, the probability of violating a constraint decreases as more submissions are made, indicating that the student is learning as more practice is being carried out with the system. The initial probability of errors (constraint violation) is .37; this drops 43% after 17 attempts, down to .16. This decrease is approximated by the power curve overlayed on the figure, which has a close fit to the power curve, with an R^2 value of 0.94.

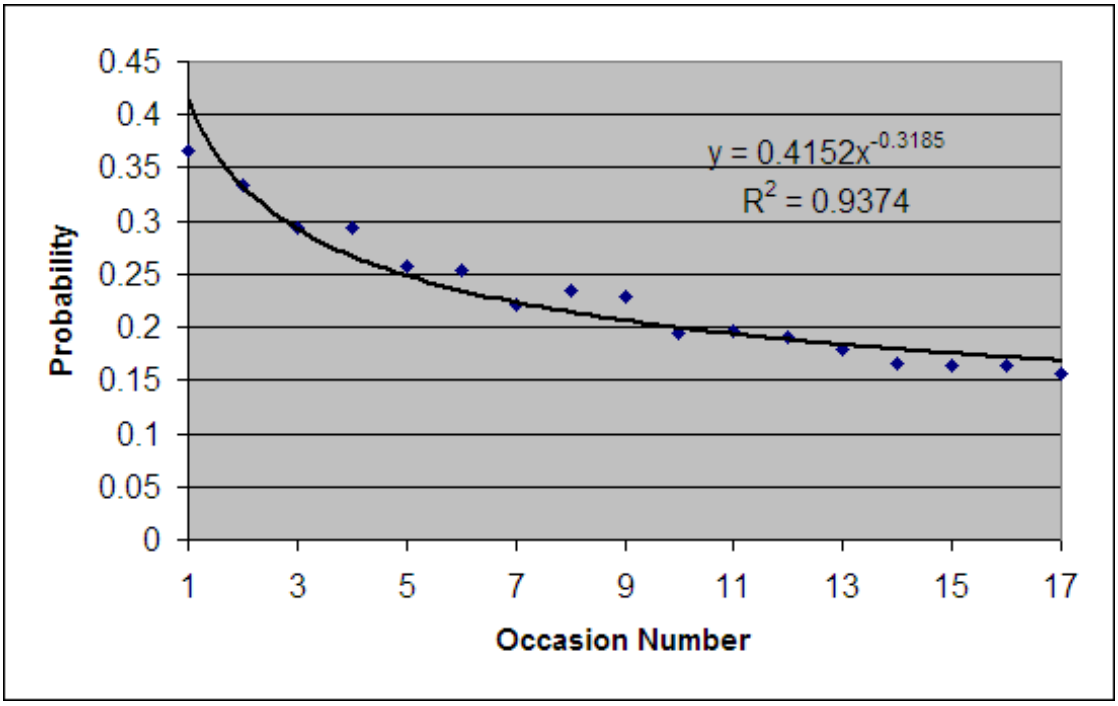

Figure 6.1: Probability of violating a constraint as a function of the occasion when that constraint was relevant, averaged over the participants in the experimental group for the 2007 evaluation

	Control	Experimental
No. of Participants	7	8
Q1 - Overall Quality - Poor to Excellent	3.4 (0.5)	3.6 (0.9)
Q2.1 - Impression - Terrible to Wonderful	3.8 (0.6)	3.5 (0.9)
Q2.2 - Impression - Difficult to Easy	3.4 (1.3)	2.9 (1)
Q2.3 - Impression - Boring to Fun	3.4 (0.5)	3.9 (0.8)
Q5 - Feedback Quality - Poor to Excellent	3.7 (1)	3.3 (1)

Table 6.3: Evaluation 2007: Mean scores from the questionnaire (Scale from 1 to 5) (s.d. given in parentheses)

Subjective Analysis

All participants completed a questionnaire at the finish of the evaluation session. Table 6.3 details the mean responses to the Likert scale questions.

Overall, it seems that the addition of fine-grained feedback was not enough to improve the perception of the experimental system over the control system, possibly due to usability flaws in the feedback itself (for example, the terminology or complexity of the messages).

The free-form questionnaire answers contained many positive comments. Some participants noted that the visual structural representation made understanding the code easier, as did breaking down the code into the abstractions. On the negative side, some comments noted a lack of flexibility for some problems, as well as some error messages being confusing. These issues were addressed after the completion of this study.

6.3 Evaluation Study 2 (August 2008)

Due to the small number of participants in the first evaluation, in August 2008 we ran the second full evaluation study. Once again the students were first-year Computer Science who were enrolled in the second semester version of the COSC121 introductory Java

programming course at the University of Canterbury. In order to obtain more participants, we ran the evaluation during normal lab times as opposed to asking for volunteers outside these times. Even though the evaluation was run during scheduled labs, participation was still essentially voluntary; there were no repercussions for students who did not attend the evaluation sessions.

For this study, the maximum session length was 110 minutes, which is the length of a lab session. This includes the initial session administration and setup, followed by a pretest, then a maximum of 90 minutes interaction time, followed finally by a posttest and a questionnaire. The data collected in the study was again anonymous; participants were given a random, non-identifying usercode to log into the system, and to link the tutor, pretest, posttest, and questionnaire data together. The participants were asked to hold onto the usercode, so they could ask us to remove their data at any time from the experiment.

In the end, despite our attempts to maximise the number of participants, only 26 students participated in the study. This was more than the first evaluation, though still many fewer than the number of students enrolled in the course (> 100).

The study was run in the 6th week of lectures, where students had covered the following material in lectures: 'Strings', 'If Statements', 'Object Interaction', and 'Loops'. Students had not yet had time to practice Loops in labs. In comparison with the first evaluation, the students had covered an extra week of material.

6.3.1 Experimental Design

As with the first evaluation, the experiment was designed to test the hypothesis that our system was more effective than a classroom setting. The definition of a classroom setting to us is that a student is given a programming problem, for example on the board,

after which they would try to answer it on paper, then once the student believed they were finished the teacher would give them the answer, and the students would compare that solution and their solution themselves.

For this evaluation, two problems were removed from the problem set due to issues with the existing constraints unable to handle their large solution spaces, and five new problems were added that covered the extended curriculum; this took the total set to 11 problems. This included 3 display problems, 4 predicate problems, and 4 loop problems.

The differences between the experimental and the control versions were identical to the previous study. Participants in the experimental group were given a version of the system with both concept level and full-solution feedback, whereas the control group participants were given a version with only full-solution feedback.

Pre and Posttests

The tests were modified slightly from the previous study to include the new material covered in lectures; each test still had 5 problems (3 short answer and 2 multichoice), and can be found in the appendices.

6.3.2 Procedure

The procedure was very similar to the previous study. Two sessions were held during the week of evaluation, one for each regular lab timeslot. Both sessions were held in the same lab. All participants carried out the evaluation using the Firefox 2.0 web browser and Fedora Core 8 operating system.

Due to having many students running the evaluation in parallel, and some students arriving at different times, explaining how to use the system to all students manually

would not be feasible; instead, an online introduction was written that explained how to use the system. A student would read this the first time they logged into J-LATTE.

The students used the system for a maximum of 90 minutes.

6.3.3 Results and Analysis

System Interaction

	Control	Experimental	Significant
No. of participants	10	14	
System interaction time (mins)	45:00 (17:58)	62:21 (18:37)	p = 0.02
No. of attempted problems	9.5 (2.2)	7.8 (2.3)	p = 0.04
No. of solved problems	2 (0.9)	5.6 (2.6)	p = 0.0002
Pretest scores	2.6 (0.9)	2.5 (1)	n.s.

Table 6.4: Evaluation 2008: System interaction statistics (s.d. given in parentheses)

Table 6.4 illustrates the data for the students' interaction with the system. Altogether there were 26 participants, but 2 participants were removed from the calculation of these results due to their interaction time being less than 10 minutes.

The significant ($t = 2.3$, $p = 0.02$, d.f. $= 20$) discrepancy in the total system interaction time between the groups can be explained by the nature of the feedback each group received. Because the control group was only able to receive the full solution, the control participant would be unable to increase their knowledge through the course of a single problem (as there was no intermediate feedback). This would mean that a control participant would be more inclined to become either stuck on a problem, or believe that their solution was correct before asking for the full solution (and therefore ending the interaction with that problem). Such a behaviour contrasts with a participant from the experimental group, who would extend their interaction time by asking for incremental feedback. Therefore, the control participant would spend less time on each problem,

and complete all the problems sooner. This behaviour would also explain the greater value for number of attempted problems, which was also significant ($t = 1.86$, $p = 0.04$, d.f. = 20).

We also found a significant difference between the number of solved problems ($t = 4.19$, $p = 0.0002$, d.f. = 22). The number of solved problems also can be explained by the lack of incremental feedback with the control group. Given the complexity of the programming domain, anything other than trivial problems would be difficult for novices to provide correct solutions for without fine-grained guidance. It would be expected that a participant may be able to easily solve the first 2 or 3 problems (i.e. single-line display problems), and then be less successful at creating perfect solutions first-time for more complex problems; the difference between the control and experimental groups is that the control only has the 'first-time'. The low standard deviation for the control group also lends weight to the argument that it is a small set of problems that can be solved without help.

Pre and posttest results are shown in Table 6.5. 4 of the participants from the control group did not submit posttests and therefore were not included in the table, in addition to the two participants with low interaction time. We found no significant difference between the pretest results of the two groups, indicating that the groups are comparable.

The improvement between the pretest and posttest for the experimental group was not significant, but the control group did show a significant difference ($t = 2.5$, $p = 0.03$, d.f. = 5). Also, the gains between the control and experimental groups (a raw gain of .83 to .29), seems concerning, and indicates that maybe the concept-level feedback was not as helpful as it should have been. Unfortunately, due to the small number of participants, the data is insignificant for drawing any solid conclusions.

Analysis of the mastery of constraints gives a similar result to the 2007 evaluation. The plot in Figure 6.2 shows that the probability of violating a constraint decreases

	#	**Pretest**	**Posttest**	Gain	Significant
Control	6	3 (0.6)	3.8 (0.9)	0.8 (0.8)	p = 0.03
Experimental	14	2.5 (1)	2.8 (1.2)	0.3 (0.9)	n.s.
Significant		n.s.	n/a	n.s.	

Table 6.5: Evaluation 2008: Pretest and posttest scores (s.d. given in parentheses)

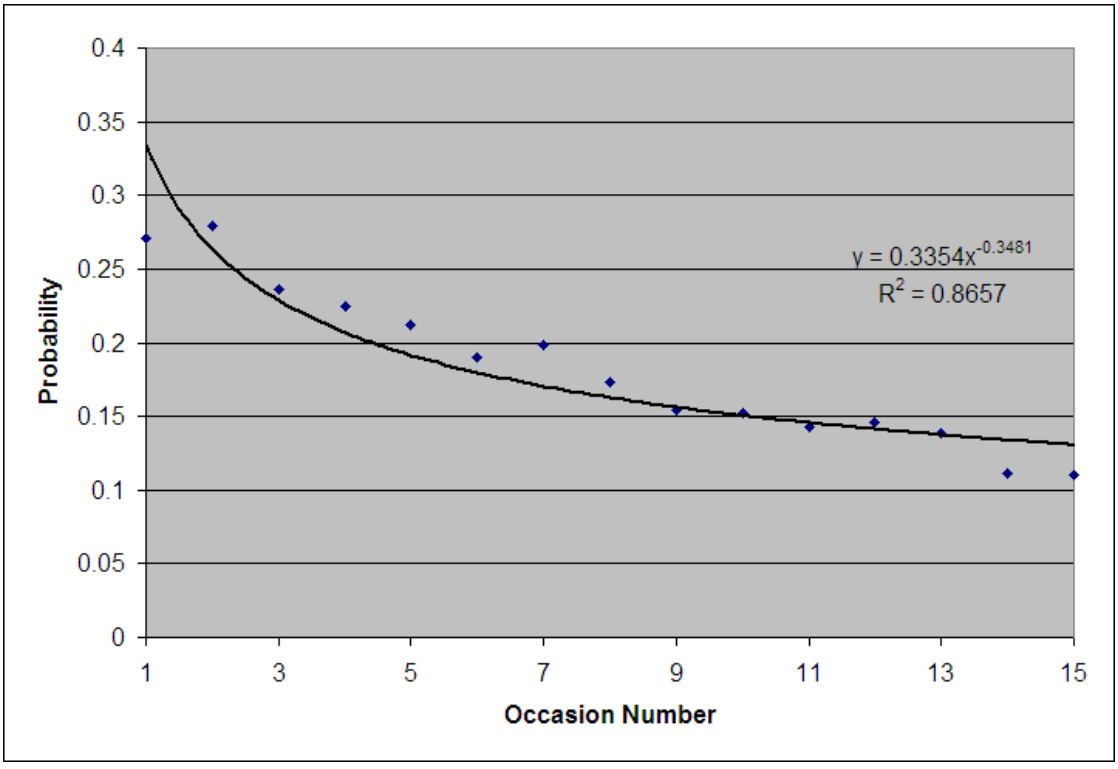

Figure 6.2: Probability of violating a constraint as a function of the occasion when that constraint was relevant, averaged over the participants in the experimental group for the 2008 evaluation

	Control	Experimental
No. of Participants	6	16
Q1 - Overall Quality - Poor to Excellent	3.5 (0.6)	3.5 (0.9)
Q2.1 - Impression - Terrible to Wonderful	3.2 (0.5)	3.3 (0.9)
Q2.2 - Impression - Difficult to Easy	3.2 (1.1)	3 (0.9)
Q2.3 - Impression - Boring to Fun	3.6 (0.7)	3.4 (0.9)
Q5 - Feedback Quality - Poor to Excellent	2.3 (0.6)	3.7 (1.1)

Table 6.6: Evaluation 2008: Mean scores from the questionnaire (Scale from 1 to 5) (s.d. given in parentheses)

as more submissions are made, once again indicating that the student is learning as more practice is being carried out with the system. The initial probability of constraint violation is .27; this probability drops 40% after 15 attempts, down to .11. This decrease is approximated by the power curve overlayed on the figure, which has a close fit to the power curve, with an R^2 value of 0.87.

There is a high correlation of 0.78 between the number of constraints learnt and the number of feedback messages, which indicates that the learning of constraints is helped by the messages received, which is to be expected. There is also a positive correlation of 0.55 between the constraints learnt and the number of solved problems.

Subjective Analysis

All participants, except for 4 members from the control group who left early, completed a questionnaire. Table 6.6 details the mean responses to the Likert scale questions.

Despite this otherwise similar impression, the response for feedback quality indicates that the participants preferred the concept-level feedback over the full solution.

CHAPTER 7

Conclusions

This thesis has presented the design and implementation of J-LATTE, an intelligent tutoring system designed to aid students in learning programming. The effectiveness of J-LATTE was evaluated through a pilot study and two evaluation studies. The results of these studies showed that the students did learn from the system.

Firstly, we outline the system itself, followed by a summary of the results of the evaluations, and finally a discussion of future work.

7.1 J-LATTE

J-LATTE is an intelligent tutoring system, designed to give novice programmers an environment to improve their Java programming skills through practice by problem-solving. Learning is more effective than problem solving in a pure programming environment, as feedback is given on the student's solution submissions to various problems. These problems represent various small-scale programming scenarios that one may encounter in a real-world programming task. J-LATTE is unique in that it breaks the interaction into two modes: concept mode, where language constructs are represented

using tiles, and coding mode, where a student may enter actual Java code to satisfy problem requirements.

J-LATTE is implemented in Allegro Common Lisp (on the server side) and XHTML and JavaScript (on the client side). The system is modular, with components that provide functionality for student modelling, and providing interactions with students via an interface. The student modeller uses constraint-based modelling to record the student's knowledge state, by recording the history of the constraints that the student has satisfied and violated. Each constraint represents a concept in the programming domain; therefore a violated constraint suggests that the student does not understand that concept. These constraints are used to evaluate solutions that are passed into the system. The conditions of each constraint are run against the solution - if they all pass, then the student's solution is correct, otherwise there are errors in the solution. Finally, the interface provides a way for the student to interact with the system, a large part of which is forming solutions (using tiles and entering code). The interface also allows the system to interact with the user, by displaying feedback to solutions.

The knowledge base of J-LATTE consists of 89 constraints (44 syntax, 43 semantic, and 2 style). Also, the final system contained 11 problems from a total of 3 problem types. The constraints and problems were developed through analysis of the Java language, observation of novice programmers working through programming tasks, and discussions with Java educators.

7.2 Evaluation

To gauge the effectiveness of J-LATTE, three studies were performed; a pilot study and two full evaluations. The goal of the pilot study was only to evaluate the students' response to the system, rather than evaluate how effective the system was, so no data on

concepts learnt was recorded. Using the think-aloud data gathered from the students, as well as an informal post-study interview, we were able to determine some issues with interaction and quality of feedback, that were fixed before the following evaluation.

The first full evaluation study was performed in a laboratory environment, with volunteer students. Students were given pre- and post-tests to analyse the learning gain, as well as a questionnaire to gather their subjective responses. Student models were also recorded by the system itself, as another method of analysing learning gain. The results of this study showed a significant difference between the pre-test and post-test of the experimental group, which indicated that the students learnt from the tutor. Also, the logs showed that the probability of constraint violation decreased significantly over time, indicating that the students were learning constraints. Most questionnaire comments were very positive about the system, and the negative comments were related to issues that could be (and were) addressed before the next evaluation.

The second full evaluation study shared similar conditions with the first full evaluation. The main difference was that the evaluation was run during regular laboratory times, in an attempt to gain a larger number of participants; also, more problems and constraints were added to the system. The results of this evaluation showed no significant difference between the pre- and post-test scores for the experimental group; still, the participant numbers were quite low, so it would be difficult to gain conclusive results. Once again though, the logs pointed towards a decrease in the probability of constraint violation over time for the experimental group, which indicates a mastery of constraints.

The research presented in this thesis further demonstrates the capabilities of CBM by successfully applying the method to a complex ill-defined domain such as programming. The results from the evaluations were inconclusive due to low participant numbers, but in the first study the indications were that the CBM feedback in such a domain

increased the learning of the experimental students over the control. Also, our unique concept/coding split approach showed positive results, with students being very receptive to being able to form solutions at a concept level before moving into coding.

7.3 Further Work

There are several avenues for further research, and for improving the J-LATTE system. Currently, the system only deals with problems that require solutions of moderate complexity. An interesting exercise would be extending the system to handle more complex problems. This would require an extension in three directions; the design of further problems, the extension (and possible revision) of the ideal solution language, and the addition of more semantic constraints. If the new problems covered more Java language constructs (e.g. such as 'Try-Catch-Finally' statements), then more syntax constraints would have to be written as well.

A related topic is the modification of the tiles. The comments in the questionnaires indicated that the participants enjoyed the scaffolding that the tiles gave. One experiment we could try is to modify the abstraction of the tiles to represent something other than straight Java blocks and statements (for example a multi-level pseudo-code representation). We could have tiles that operate at a finer level than the statements, so that a student can be more precise with their intentions.

Due to the small numbers of participants in our experiments, further evaluation of the system with a larger number of participants would be preferable to acquire a more conclusive result. One possibility would be to evaluate the system at other institutions, or with non-university programming courses.

The system does not currently have an automatic problem-selection facility - the student chooses their own way through the problems (though we inform the student to

move through the problems sequentially). We are already recording a student model for each user, so implementing this would be a case of identifying the concepts associated with each problem, using the student model to identify which concepts the student is having issues with, and using this data to choose a problem where this concept is required.

References

Aleven, V. and Koedinger, K. R. (2000). Limitations of student control: Do students know when they need help? In *ITS '00: Proceedings of the 5th International Conference on Intelligent Tutoring Systems*, pages 292–303, London, UK. Springer-Verlag.

Allen, E., Cartwright, R., and Stoler, B. (2002). DrJava: a lightweight pedagogic environment for Java. *Proceedings of the 33rd SIGCSE technical symposium on Computer science education*, pages 137–141.

Anderson, J. (1996). ACT: A simple theory of complex cognition. *American Psychologist*, 51(4):355–365.

Anderson, J. and Reiser, B. (1985). The LISP tutor. *Byte*, 10(4):159–175.

Anderson, J. R., Corbett, A. T., Koedinger, K., and Pelletier, R. (1995). Cognitive tutors: Lessons learned. *The Journal of Learning Sciences*, 4:167–207.

Baghaei, N. and Mitrovic, A. (2006). A constraint-based collaborative environment for learning uml class diagrams. In *ITS2006. 2006*, pages 176–186. Springer.

Baghaei, N., Mitrovic, A., and Irwin, W. (2005). A constraint-based tutor for learning object-oriented analysis and design using uml. In *ICCE 2005*, pages pp.11–18.

Baghaei, N., Mitrovic, A., and Irwin, W. (2007). Supporting collaborative learning and problem-solving in a constraint-based cscl environment for uml class diagrams.

International Journal of Computer-Supported Collaborative Learning, 2(2-3):159–190.

Baker, R. S., Corbett, A. T., Koedinger, K. R., and Roll, I. (2005). Detecting when students game the system, across tutor subjects and classroom cohorts. In *User Modeling*, pages 220–224.

Beck, J., Stern, M., and Haugsjaa, E. (1996). Applications of AI in education. *Crossroads*, 3(1):11–15.

Bloom, B. S. (1984). The 2 sigma problem: The search for methods of group instruction as effective as one-to-one tutoring. *Educational Researcher*, 13(6):4–16.

Butz, C., Hua, S., and Maguire, R. (2004). A Web-based Intelligent Tutoring System for Computer Programming. *Proceedings of the Web Intelligence, IEEE/WIC/ACM International Conference on (WI'04)-Volume 00*, pages 159–165.

Daly, C. and Horgan, J. (2004). An automated learning system for Java programming. *IEEE Transactions on Education*, 47(1):10–17.

Fernandes, E. and Kumar, A. N. (2004). A tutor on scope for the programming languages course. In *SIGCSE '04: Proceedings of the 35th SIGCSE technical symposium on Computer science education*, pages 90–93, New York, NY, USA. ACM.

Forgy, C. L. (1982). Rete: a fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19:17–37.

Gertner, A. and VanLehn, K. (2000). Andes: A Coached Problem Solving Environment for Physics. In Gauthier, G., Frasson, C., and VanLehn, K., editors, *Intelligent Tutoring Systems: 5th International Conference*, pages 131–142. Berlin: Springer.

Gosling, J., Joy, B., Steele, G., and Bracha, G. (2000). *Java Language Specification, Second Edition: The Java Series*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Greer, J. and McCalla, G. (1994). *Student modelling: the key to individualized knowledge-based instruction*. Springer-Verlag, New York.

Irwin, W., Cook, C., and Churcher, N. (2005). Parsing and semantic modelling for software engineering applications. *Software Engineering Conference, 2005. Proceedings. 2005 Australian*, pages 180–189.

Johnson, W. L. and Soloway, E. (1984). PROUST: Knowledge-based program understanding. *Proceedings of the 7th international conference on Software engineering*, pages 369–380.

King, J. C. (1976). Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394.

Koedinger, K. R., Anderson, J. R., Hadley, W. H., and Mark, M. (1997). Intelligent tutoring goes to school in the big city. *International Journal of Artificial Intelligence in Education*, 8:30–43.

Kostadinov, R. and Kumar, A. (2003). A tutor for learning encapsulation in c++ classes. In Lassner, D. and McNaught, C., editors, *Proceedings of World Conference on Educational Multimedia, Hypermedia and Telecommunications 2003*, pages 1311–1314, Honolulu, Hawaii, USA. AACE.

Kumar, A. (2004). Web-based tutors for learning programming in C++/Java. *Proceedings of the 9th annual SIGCSE conference on Innovation and technology in computer science education*, pages 266–266.

Kumar, A. and Dancik, G. (2003). A tutor for counter-controlled loop concepts and its evaluation. *Frontiers in Education, 2003. FIE 2003. 33rd Annual*, 1:T3C–7–12 Vol.1.

Martin, B. and Mitrovic, A. (2002). Authoring web-based tutoring systems with WETAS. In *Proceedings of the International conference on computers in education*, pages 183–187, Auckland.

Mitrovic, A., Koedinger, K. R., and Martin, B. (2003). A comparative analysis of cognitive tutoring and constraint-based modelling. In *Proceedings of the 9th International Conference on User Modeling (UM2003)*, pages 313–322. Springer-Verlag.

Mitrovic, A. and Martin, B. (2003). Scaffolding and fading problem selection in SQL-Tutor. In *Proceedings of the 11th International Conference on Artificial Intelligence in Education (AIED 2003)*, pages 479–481. IOS Press.

Mitrovic, A. and Ohlsson, S. (1999). Evaluation of a Constraint-Based Tutor for a Database Language. *Int. J. Artificial Intelligence in Education*, 10(3-4):238–256.

Ohlsson, S. (1994). Constraint-based Student Modeling. In *Student Modeling: the Key to Individualized Knowledge-based Instruction*, pages 167–189. Springer-Verlag.

Ohlsson, S. (1996). Learning from performance errors. *Psychological Review*, 103(2):241–262.

Self, J. (1990). Bypassing the intractable problem of student modelling. In *Intelligent Tutoring Systems: at the Crossroads of Artificial Intelligence and Education*, pages 107–123. Norwood: Ablex.

Shah, H. and Kumar, A. N. (2002). A tutoring system for parameter passing in programming languages. *SIGCSE Bull.*, 34(3):170–174.

Suraweera, P. and Mitrovic, A. (2004). An Intelligent Tutoring System for Entity Relationship Modeling. *Int. J. Artificial Intelligence in Education*, 14(3-4):375–417.

Suraweera, P., Mitrovic, A., and Martin, B. (2005). A knowledge acquisition system for constraint-based intelligent tutoring systems. In Looi, C.-K., McCalla, G., Bredeweg, B., and Breuker, J., editors, *Proc. Artificial Intelligence in Education AIED 2005*, pages 638–645. IOS Press.

Sykes, E. R. and Franek, F. (2003). An Intelligent Tutoring System Prototype for Learning to Program Java. In *3rd IEEE International Conference on Advanced Learning Technologies, Athens, Greece*, pages 485–486.

Sykes, E. R. and Franek, F. (2004). Inside the Java Intelligent Tutoring System Prototype: Parsing Student Code Submissions with Intent Recognition. In *IASTED International Conference on Computers and Advanced Technology in Education*, pages 613–618, Innsbruck, Austria.

Taylor, R. (1980). *The Computer in the school : tutor, tool, tutee / edited by Robert Taylor ; [photos. by Louis Forsdale and Robert P. Taylor]*. Teachers College Press, New York.

Van Haaster, K. and Hagan, D. (2004). Teaching and Learning with BlueJ: an Evaluation of a Pedagogical Tool. *Information Science+ Information Technology Education Joint Conference*.

Weerasinghe, A. and Mitrovic, A. (2002). Enhancing learning through self-explanation. In *ICCE '02: Proceedings of the International Conference on Computers in Education*, pages 244–248, Washington, DC, USA. IEEE Computer Society.

Wei, F., Moritz, S. H., Parvez, S. M., and Blank, G. D. (2005). A student model for object-oriented design and programming. *Journal of Computing Sciences in Colleges*, 20(5):260–273.

Woolf, B. and Cunningham, P. (1987). Building a community memory for intelligent tutoring systems. In *Proc. of AAAI-87*, pages 82–87, Seattle, WA.

Zia, H., Durrani, Q. S., Farrakh, R. A., Riaz, A., and Ahmed, F. (1999). CITS - C++ Intelligent Tutoring System: A Domain Independent User Centered Curriculum Approach. In *International Joint Conference on Artificial Intelligence*.

Appendices

In this part of our thesis we append the following from the evaluation studies:

- Information sheet and Consent form
- Pre and Post Tests
- Questionnaire

Also included is a paper presented at NZCSRSC 2007.

Appendix A

Information Sheet

The following page contains the information sheet used in both evaluation studies.

J-LATTE System Evaluation J-LATTE Usercode: _____

Thank you for participating in this evaluation study. The aim of the study is to investigate the effectiveness of J-LATTE (Java-Language Acquisition Tile Tutoring Environment). You are expected to work individually, solving problems at your own pace. Before and after you use the system itself, you will be required to solve some programming-related problems on paper. After using the system for around 90 minutes (or until all J-LATTE problems are completed), you will also be required to answer some questions about your view of the system itself.

This is not assessing your competence or intelligence in any way. All the data reported on this study will be anonymous. You are free to stop the session at any time, and also to require that your session is not used in the study. If you wish to withdraw your session at a later date, then please contact Jay Holland (see below), and supply the J-LATTE usercode (at the top of this page) that you are given to access the system.

This project is carried out by Jay Holland, who is an M.Sc student at the Department of Computer Science and Software Engineering, University of Canterbury, and is supervised by Dr Antonija Mitrovic and Dr Brent Martin. He can be contacted through email at jah130@student.canterbury.ac.nz. He will be happy to discuss any concerns you may have about participating in the project.

Please keep this sheet for your future reference.

Appendix B

Consent Form

The following page contains the consent form used in both evaluation studies.

Consent Form for J-LATTE System Evaluation

I have read and understood the description of the above-named project. On this basis, I agree to participate in this project, and I consent on the publication of the results of the project with the understanding that anonymity will be preserved. I understand also that I may at any time withdraw from the project, including withdrawal of any information I have provided.

Signed: …… Date: ……

Name: …….

Appendix C

Pre and Post Tests

The following four pages contain the evaluation pre- and post-tests in the following order:

- Evaluation 2007 - Test 1
- Evaluation 2007 - Test 2
- Evaluation 2008 - Test 1
- Evaluation 2008 - Test 2

Question 1:
At the end of this code, what value will the variable 'myVar2' be set to?

```
String myVar = "hello";
int myVar2;
myVar2 = myVar.indexOf("d");
```

Answer: _____

Question 2:
The method within this class will not compile for three reasons. What are they?

```
public class MyClass {

        public void doSomething(myArg){
                int x=7;
                x++32;
                return x;
        }
}
```

Answer: _____

Question 3:
Based on the information in this method, what type would the 'myVar' variable be? (i.e. what keyword would go in the blank space before 'myVar').

```
        public void myMethod (int x ){
                _____ myVar;
                myVar=(x<7);
                if (myVar){
                 doSomethingElse( );
                }
        }
```

Answer: _____

Question 4:
If x were set to 30, which comment would the program reach?
a) the first one, b) the second one, c) none, d) both

```
        if (x<30){
                //comment 1
          }
        if (x>=45){
                //comment 2
          }
```

Answer: _____

Question 5:
Write a line of code assigning the result of whether 'x' and 'y' are equal strings, to the variable myResult (assume that myResult has not been declared before, so you'll need to specify the type).

Answer: _____

Question 1:
At the end of this code, will myVar2 be set to ‘true’ or ‘false’?

```
String myVar = “hello”;
boolean myVar2;
myVar2 = (myVar == “hello”);
```

Answer: _____

Question 2:
The method within this class will not compile for three reasons. What are they?

```
public class MyClass {

        public String doAction(in myArg){
                int x=26;
                x=myArg
        }
}
```

Answer: _____

Question 3:
Based on the information in this method, what type would the ‘myVar’ variable be? (i.e. what keyword would go in the blank space before ‘myVar’).

```
        public void myMethod (){
                _____ myVar;
                myVar=7+32;
        }
```

Answer: _____

Question 4:
If x were set to 30, which comment would the program reach?
a) the first one, b) the second one, c) none, d) both

```
        if (x!=30){
                //comment 1
         }
        if (x!=25){
                //comment 2
         }
```

Answer: _____

Question 5:
Write a line of code assigning the sum of the number variables ‘x’ and ‘y’, to the variable mySum (assume that mySum has not been declared before, so you’ll need to specify the type)

Answer: _____

Question 1:
At the end of this code, what value will the variable ‘myVar2’ be set to?

```
String myVar = “hello”;
int myVar2;
myVar2 = myVar.indexOf(‘d’);
```

Answer: _____

Question 2:
The method within this class will not compile for three reasons. What are they?

```
public class MyClass {

        public void doSomething(myArg){
                int x=7;
                x++32;
                return x;
        }
}
```

Answer: _____

Question 3:
Based on the information in this method, what type would the ‘myVar’ variable be? (i.e. what keyword would go in the blank space before ‘myVar’).

```
        public void myMethod (int x ){
                _____ myVar;
                myVar=(x<7);
                if (myVar){
                 doSomethingElse( );
                }
        }
```

Answer: _____

Question 4:
If x were set to 30, which comment would the program reach?
a) the first one, b) the second one, c) none, d) both

```
        if (x<30){
                //comment 1
          }
        if (x>=45){
                //comment 2
          }
```

Answer: _____

Question 5:
In a ‘for’ loop heading, what is the purpose of the third section inside the parentheses?

Answer: _____

Question 1:
At the end of this code, will myVar2 be set to ‘true’ or ‘false’?

```
String myVar = “hello”;
boolean myVar2;
myVar2 = (myVar == “hello”);
```

Answer: _____

Question 2:
The method within this class will not compile for three reasons. What are they?

```
public class MyClass {

        public String doAction(in myArg){
                int x=26;
                x=myArg
        }
}
```

Answer: _____

Question 3:
Based on the information in this method, what type would the ‘myVar’ variable be? (i.e. what keyword would go in the blank space before ‘myVar’).

```
        public void myMethod (){
                _____ myVar;
                myVar=7+32;
        }
```

Answer: _____

Question 4:
If x were set to 30, which comment would the program reach?
a) the first one, b) the second one, c) none, d) both

```
        if (x!=30){
                //comment 1
         }
        if (x!=25){
                //comment 2
         }
```

Answer: _____

Question 5:
In a ‘for’ loop heading, what is the purpose of the second section inside the parentheses?

Answer: _____

Appendix D

Questionnaire

The following page contains the questionnaire used in both evaluation studies.

Questionnaire

J-LATTE Usercode: _____

1. How would you rate the overall quality of J-LATTE?

1 ---------- 2 ---------- 3 ---------- 4 ---------- 5
Poor Excellent

2. Rate your impression of J-LATTE:

1 ---------- 2 ---------- 3 ---------- 4 ---------- 5
Terrible Wonderful

1 ---------- 2 ---------- 3 ---------- 4 ---------- 5
Difficult Easy

1 ---------- 2 ---------- 3 ---------- 4 ---------- 5
Boring Fun

3. What did you like about J-LATTE?

_____
_____
_____
_____
_____

4. What changes would you like to see in J-LATTE?

_____
_____
_____
_____
_____

5. How would you rate the overall quality of the feedback from J-LATTE?

1 ---------- 2 ---------- 3 ---------- 4 ---------- 5 I haven't
Poor Excellent used it

6. What changes would you like to see in J-LATTE's feedback?

_____
_____
_____
_____
_____

Other comments about J-LATTE

_____
_____
_____
_____
_____

Appendix E

NZCSRSC 2007 Paper

The following paper was presented at the 5th New Zealand Computer Science Research Student Conference that was held at Waikato University in April 2007.

A Constraint-Based Intelligent Tutoring System for the Java Programming Language

Jay Holland, Antonija Mitrovic

Intelligent Computer Tutoring Group,
Computer Science Department, University of Canterbury
Private Bag 4800, Christchurch, New Zealand
E-mail: {jah130,tanja}@cosc.canterbury.ac.nz

Abstract. This paper presents the design and implementation of Java-ITS, a constraint-based intelligent tutoring system for teaching the Java programming language. In order to learn programming, a student must acquire new cognitive skills, which when coupled with having to also learn the syntax of a particular programming language (necessary to apply a practical context to this skill), can make the process overwhelming. Even if a student can understand programming at a micro-level, to be a better programmer they must be aware of the overall design and context of a program, a useful skill that is often an afterthought. The goal of our project is to make the process of gaining programming skill both accessible through smoothing the learning curve, and relevant (from a practical perspective), such that transfer problems are reduced.

1 Introduction

Acquisition of computer programming skill is a core component of the Computer Science curriculum, a fact reflected in the many first-year tertiary prescriptions that require a student to undertake some kind of programming course. There are many aspects to programming theory, such as program control-flow and scope, and this variety can make it difficult for students already lacking a suitable information technology background [1]. It is generally accepted that the best way to introduce these ideas is through the teaching of a specific language. The Java programming language provides an appropriate introductory programming syllabus. Due to its low-level abstractions and system-independent nature, the student is able to concentrate more on the general programming concepts rather than system idiosyncrasies.

Although programming courses tend to have material taught in lectures, most of the learning reinforcement takes place in laboratories, where practical tasks are carried out. An increasingly popular and effective way of improving student learning is through Intelligent Tutoring Systems (ITSs), which enhance learning by providing feedback personalised to a student. These have been shown to be effective for many different disciplines and areas, including mathematics [2], physics [3] and database design [4]. The Java-ITS system, which is part of a master's of science project, is our attempt to teach the Java language to students, through a tutor that utilizes the constraint-based modelling (CBM) methodology [5]. Section 2 presents related work, fol-

lowed by the discussion of architecture and design decisions behind the system in Section 3. We conclude the paper by presenting future work in the final section.

2 Related Work

2.1 Intelligent Tutoring Systems

Personal tutoring is one of the most effective ways of enhancing learning. Due to growing populations and the complexities of some fields, personal tutors are not always readily available, whereas computers are becoming more and more commonplace. From the early days of computing Computer-Aided Instruction (CAI) has been a prominent field for research into how to achieve the same effectiveness as a personal human tutor. The first systems were primitive in terms of how they reacted to the students' behaviour; there was little or no adaptation to the student's progress, and they generally just followed a linear script. This changed with the advent of ITSs, which calculated the proficiency of students in various concepts related to the field of the tutor, and used this information to personalise the tutoring. This skill-tracking is known as *student modelling*. An interdisciplinary field, ITS theory draws from psychology and education as well as computer science, as we try to model and understand the cognitive process.

2.2 Constraint-Based Modeling

CBM handles student modelling by representing all domain knowledge in the form of state constraints. Each constraint is an ordered pair made up of a relevance condition and a satisfaction condition. For a given solution any relevant constraints (i.e. constraints whose relevance conditions are met by the students solution) must be satisfied to have the solution be evaluated as correct. Any constraint violations indicate an error in the solution; this in turn indicates the student has an incorrect understanding of the domain knowledge of each violated constraint.

2.3 Intelligent Programming Tutors

Very few ITSs teach general programming skills through free-form coding. Several ITSs focus on a single skill and tailor the interface to the particular skill; e.g Kumar's set of C++ and Java tutors [6], which teach, each in a separate system, expression evaluation, for loops, and C++ pointers, amongst other topics.

In terms of programming as a ‘coding’ activity, one of the most popular ITSs has been the Carnegie-Mellon LISP tutor [7]; it was also one of the first programming ITSs. A model-tracing tutor, it has provided a good starting point for other programming-tutor research. An evaluation of the system showed that its effectiveness approached that of a human tutor; on average, students covered the entire course curriculum in 15 hours, which was only 3.6 hours worse than the average time taken for

the students to complete the material with a human tutor (11.4), and 11.5 hours better than learning without either (26.5). Covering the material in a classroom setting takes over 40 hours.

The Java Intelligent Tutoring System (JITS) [8] is another Java tutor to allow free-form coding, albeit with no design section. The system is built around the core of the "intent-recognition algorithm" [9]. Several strategies are implemented to attempt to predict what the student was intending to accomplish with the code. One such strategy, the "Syntax Error Correction Strategy", finds unrecognisable tokens in a student's submission, and reverses possible error transformations such as the replacement of a symbol by another symbol, or the insertion of an extraneous symbol, to attempt to correct to code; if a more meaningful code chunk is obtained, the system assumes that a syntax error is present, and that the corrected code chunk represents the true intent of the student. JITS will then initiate a dialogue with the student, which includes questions about sections of the code; for example, the system may ask "I see 'intt'. Do you mean the keyword 'int'?" The dialogue continues until the code is completely correct.

3 The Java-ITS System

Java-ITS is an intelligent tutoring system where students can form solutions to various Java programming problems, and receive feedback on their solutions. The curriculum the system supports is a subset of the Java programming language; as the complete Java domain is vast, incorporating all the domain knowledge into a single tutor, while not impossible, would require an immense amount of effort to validate that everything was correctly implemented. By working with a subset, we are able to still give an appropriate learning experience, yet maintain validity of the concepts taught. If there is a need to extend the tutor in the future to cover more of the domain, more concepts can be incrementally added and validated by extending the domain model and problem set. As the target audience of the tutor is novice programmers, the curriculum begins with the most elementary of concepts and follows a typical tertiary course progression. By working through all the problems in the system, the student should gain a good understanding of all programming concepts up to and including loops.

3.1 Architecture

The system architecture of Java-ITS adheres closely to the architecture of other constraint-based tutors, as illustrated in Figure 1. All information and interaction is presented to the student through a web interface, which can be viewed in any mainstream web-browser. The session manager handles any requests from the web-server. It works as a hub, and interacts with most other parts of the system at some point. Pedagogical decisions and operations take place in the pedagogical module (PM). This receives the interactions (via the web-server and the session-manager) from the student, such as problem selection and solution submission.

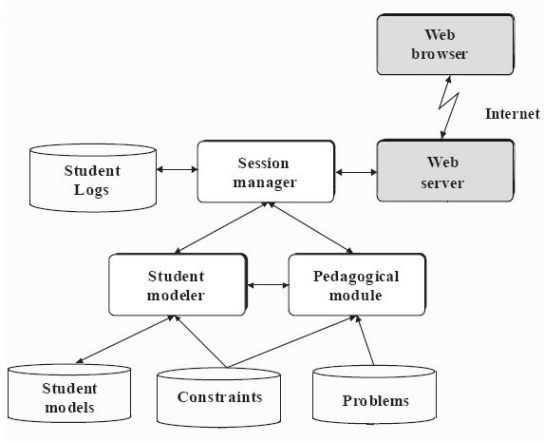

Figure 1: The Architecture of Java-ITS

The constraint store is the knowledge base of the system, and contains all the constraints that make up the domain model. The constraints can be divided into three categories: ones that examine syntactic properties, ones that examine semantic properties of a solution, and ones that examine style – although there are many ways to write a program, it is better to encourage good practice. Each constraint contains three parts: a relevance condition, a satisfaction condition, and a feedback message. The relevance and satisfaction conditions are examined during evaluation, and the feedback message is shown to the student if that constraint is violated.

In terms of the problem set, with many tutors the goal of each problem within the tutor will be similar, with a general template being used to generate further problems in the same goal set. With programming, the area is so broad that different skills are required by programming solutions to problems with vastly different outcomes that are hard to generalise; therefore, in a programming tutor, it is difficult to keep to just one form without restricting learning. To solve this issue in Java-ITS, the tutor's problems have been broken into groups, with each group containing problems of a certain type. For example, here is an example of a problem of an 'iteration' type:

"Bob has two cats, Fluffy and Whiskers. Fluffy needs to be given milk every day, but Whiskers only needs to be given milk every second day. Complete the following method, `feedCats`*, such that it iterates over given number of days, and milk is only given to each cat on the appropriate days."*

Each problem is presented with its own *context*. The context is a code fragment that frames a problem, and is displayed inside the solution workspace, so that the student has existing properties to work with. For example, the context could be a 'for' loop beginning and end, or a method outline (signature and braces). Often there will be variables and other methods that the student can reference in their own code, such as arguments to methods; in fact, often these variables will be mentioned directly in the problem text itself, and therefore the student will be expected to use them in some way. The context for the previous `feedCats` problem is as follows:

```
Cat whiskers;
Cat fluffy;
public void feedCats (int days) {
}
```

Each problem has a corresponding ideal-solution, which is used by the semantic constraints during evaluation to semantically validate the student's solution. It describes an abstract version of what is required from the submission, i.e. rather than explicitly specifying what design concepts and code fragments should occur in the submitted solution, it only notes the general requirements of the given problem that must manifest in the solution for the problem's tasks to be considered satisfied, such as "the solution must return this variable" or "must loop up to this value". It is essentially a formal specification of the problem statement.

The Student Modeller (SM) is responsible for maintaining the student models. It receives the list of violated and satisfied constraints from the solution evaluation, and appends this information to a student's model. The SM can also provide details relating to a model, such as how well a student knows a particular constraint; this information will be used by the pedagogical module to make problem-selection decisions.

Via the web-based interface, the student is able to perform all necessary interactions. These include operations such as problem selection, problem solving, and submitting a solution. The problem solving interface, illustrated in Figure 2, is where the student will spend most of their time. The screen is split into several panes, with four main panes being related to tutoring activities. The top pane presents the problem text to the student, while the large middle pane is the solution workspace and allows the student to form a solution to the given problem. The bottom pane contains components (tiles) to be used in solution formation, and the rightmost pane presents feedback on a student's solution.

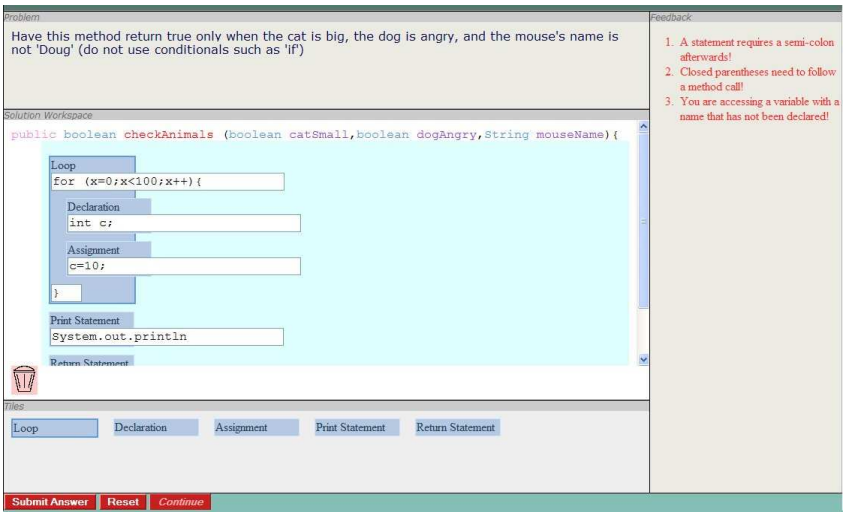

Figure 2: Problem Solving Interface (Coding Stage)

3.2 Problem-Solving Stages

The interface (and the general layout of the task) has been designed such that the student is better able to handle the complexity of a program, by first presenting the student with generic programming constructs in the form of tiles. Problem solving is done in two phases: the student firstly designs the program in terms of generalised solutions using tiles, followed by a coding stage, where he/she enters actual code fragments into the tiles.

A typical tutoring session progresses as such: The student selects a problem, and they are presented with the problem-solving interface, set to the design stage, populated with any information necessary to the task (problem text, context, solution tiles). The student then designs the solution using the tiles, and submits the solution when he/she believes they have a correct solution, or is unable to continue. The system then returns feedback to the student, revealing any errors that may exist in the solution. The student can use this feedback to try and correct the solution, and submit again. This submit/feedback loop will continue until the student correctly forms a solution, at which point the feedback will indicate that the solution is indeed correct, and the student can then move on to the coding stage. A similar process as with the design stage then takes place, albeit with code fragments instead. Once the student submits a correct solution to the coding stage, then they can move on to the next problem.

The design stage is characterised by the use of *tiles*. Each tile, housed in the tile pane below the solution workspace, represents a different abstract programming construct, such as a loop or a variable declaration. Tiles can be categorised into two groups: *statement* tiles, which are equivalent to a line of code in Java, and *block* tiles, which also act as containers for other tiles (a method is an example of a block tile).

To design solutions, students drag-and-drop tiles from the tile pane and place them inside the solution workspace input areas, and possibly inside other tiles. Using the tile pane is a ‘cloning’ action - the tiles will not disappear from the tile pane once used, and the student is free to make as many copies as needed. Tiles in the solution workspace can be deleted if necessary, and once placed inside the solution, they can still be moved to other parts of the solution.

The coding stage is built upon the solution the student generated previously in the design stage. Upon entering the coding stage, the student is presented with the tile-based solution developed in the design stage, but text-entry boxes have now been inserted inside each tile (two for block tiles), and the tiles themselves are now immovable (the ability to add new tiles to the solution has also been taken away). The student must now complete these text entry boxes by entering Java code. The result of this is that the solution, once completed, will resemble a real-life code listing.

The two-stage principle is a product of the main goals of the system. The first goal was to provide accessible tutoring - as programming is a complex activity, the system would benefit the student's learning by providing a way for the student to handle that complexity, therefore it is mandatory for the student to design the solution first using tiles. By making the design task explicit, the student will have to consider the structure. The second goal was to reduce the transfer problems between the tutoring system and a real-life coding situation; therefore the system was designed such that the student would be at some stage entering code themselves, similar to using a text editor during normal programming tasks. There is evidence that suggests that although stu-

dents tend to retain more information for a purely text-entering approach, they enjoy using the system less than symbolic approaches [10]. By combining symbolic and partial free-form text, we hope to receive the benefits of both approaches. Having the symbolic approach first reduces the memory load during the coding stage, and also smoothes out the learning curve.

The two-stage approach also aids the server-side solution evaluation; by reducing the ambiguity of the student's completed solution, the amount of reasoning required by the system is reduced, which in turn simplifies the system implementation. If we were to evaluate a solution composed of only free-form text, then we would run into the same problems as a compiler would; compiler messages are often misleading due to fact that common errors include missing semicolons or braces, meaning it can be difficult to tell where a method really ends, or the boundaries of a statement. Also, the intent of the student is not always clear inside free-form text. If a student were to create a line of code that was completely syntactically incorrect, we may not be able to tell if it they were trying to write a conditional or simply make a method call; in situations like this feedback would be limited and general, and possibly inaccurate. By forcing the student to enter their code inside the tiles, we are able to decipher the intent of a statement by the tile type, and can provide more specific feedback.

3.3 Solution Evaluation

The student modeller module handles solution evaluation. Once a submission is received from the student, all constraints are evaluated over the solution in two stages. The relevance conditions are initially evaluated to deduce which constraints are relevant to the current problem and/or solution. The ones that are considered relevant then have their satisfaction conditions evaluated – any relevant constraints not passing this stage will be considered violated, which means the student has not learnt the domain concept the constraint represents, otherwise they will be considered satisfied, indicating the student does understand the concept.

The process is executed by propagating the solution through a constraint network, which is loosely based on a Rete network, in order to optimise the potentially intensive procedure. Each node in the network references part or all of a condition, and specifies which constraints the condition came from, such that the system knows which constraints to apply to the result of evaluating the node’s condition.

4 Future Work

Java-ITS has not yet been evaluated as to its effectiveness as a tutoring tool. A full evaluation is planned for early 2007, to be taken with an introductory Java programming class.

Although supporting the entire Java domain is beyond the immediate scope of this project, the system can be incrementally improved through developing more constraints, therefore extending the domain coverage. In addition, the problem set can also be increased, by either working inside a template to develop more complex problems (for example increasing the number of clauses), or developing new templates

that focus on different problem goals. If any new design concepts are introduced through new constraints and problems, then new tiles will need to be introduced into the interface. More research must be made into 'good practice' coding style in order to generate more style constraints.

The system can also be extended to support problem and feedback-level selection. Currently, the student manually selects the problem they wish to work on, but the system can be adapted to provide suggestions or making problems mandatory, depending on how well a student understands a concept; if a student is unfamiliar with the loop construct, we can suggest problems which deal lightly with loops at first, then move them on to more complex problems. With feedback-level selection, with can give varying degrees of hints, to allow the student to think more about the problem themselves before receiving more specific feedback.

References

1. Pillay, N.: Developing Intelligent Programming Tutors for Novice Programmers. Inroads - the SIGCSE Bulletin **35** (2003) 78-82
2. Singley, M.K., Anderson, J.R., Gevins, J.S., Hoffman, D.: The algebra word problem tutor. Artificial Intelligence and Education (1989) 267-275
3. Gertner, A., VanLehn, K.: Andes: A Coached Problem Solving Environment for Physics. In: VanLehn, K. (ed.): Intelligent Tutoring Systems: 5th International Conference. Berlin: Springer (2000) 131-142
4. Suraweera, P., Mitrovic, A.: An Intelligent Tutoring System for Entity Relationship Modeling. Int. J. Artificial Intelligence in Education **14** (2004) 375-417
5. Ohlsson, S.: Constraint-based Student Modeling. Student Modeling: the Key to Individualized Knowledge-based Instruction. Springer-Verlag (1994) 167-189
6. Kumar, A.: Web-based tutors for learning programming in C++/Java. Proceedings of the 9th annual SIGCSE conference on Innovation and technology in computer science education (2004) 266-266
7. Anderson, J.R., Reiser, B.J.: The LISP tutor. Byte **10** (1985) 159-175
8. Sykes, E.R., Franek, F.: An Intelligent Tutoring System Prototype for Learning to Program Java. 3rd IEEE International Conference on Advanced Learning Technologies, Athens, Greece (2003) 485-486
9. Sykes, E.R., Franek, F.: Inside the Java Intelligent Tutoring System Prototype: Parsing Student Code Submissions with Intent Recognition. IASTED International Conference on Computers and Advanced Technology in Education, Innsbruck, Austria (2004) 613-618
10. Corbett, A.T., Anderson, J.R., Fincham, J.M.: Menu selection vs. typing: effects on learning in an intelligent programming tutor. International Conference of the Learning Sciences, Evanston, IL (1991) 107-112